LoadRunner®

Creating GUI Virtual User Scripts

UNIX

Version 6.0

LoadRunner® Creating GUI Virtual User Scripts (UNIX), Version 6.0

If you have any comments or suggestions regarding this document, please send them via e-mail to documentation@mercury.co.il.

Mercury Interactive Corporation
1325 Borregas Avenue
Sunnyvale, CA 94089 USA

LRGUIUXUG6.0/03

Table of Contents

Welcome to LoadRunner

Welcome to LoadRunner, Mercury Interactive's tool for testing the performance of applications. LoadRunner stresses your application to isolate and identify potential client, network, and server bottlenecks.

LoadRunner enables you to test your system under controlled and peak load conditions. To generate load, LoadRunner runs thousands of Virtual Users that are distributed over a network. Using a minimum of hardware resources, these Virtual Users provide consistent, repeatable, and measurable load to exercise your application just as real users would. LoadRunner's in-depth reports and graphs provide the information that you need to evaluate the performance of your application.

Online Resources

LoadRunner includes the following online tools:

Read Me First provides last-minute news and information about LoadRunner.

Books Online displays the complete documentation set in PDF format. Online books can be read and printed using Adobe Acrobat Reader, which is included in the installation package. Check Mercury Interactive's Customer Support Web site for updates to LoadRunner online books. The URL for this Web site is *http://support.mercuryinteractive.com*.

LoadRunner Online Function Reference gives you online access to all of LoadRunner's functions that you can use when creating Vuser scripts, including examples of how to use the functions. Check Mercury Interactive's Customer Support Web site for updates to the *LoadRunner Online Function Reference*.

LoadRunner Context Sensitive Help provides immediate answers to questions that arise as you work with LoadRunner. It describes dialog boxes, and shows you how to perform LoadRunner tasks. To activate this help, click in a window and press F1. Check Mercury Interactive's Customer Support Web site for updates to LoadRunner help files.

Technical Support Online uses your default web browser to open Mercury Interactive's Customer Support web site. The URL for this Web site is *http://support.mercuryinteractive.com*.

Support Information presents the locations of Mercury Interactive's Customer Support web site and home page, and a list of Mercury Interactive's offices around the world.

Mercury Interactive on the Web uses your default Web browser to open Mercury Interactive's home page. The URL for this Web site is *http://www.mercuryinteractive.com*.

LoadRunner Documentation Set

LoadRunner is supplied with a set of documentation that describes how to:

- install LoadRunner
- create Vuser scripts
- use the LoadRunner Controller

Using the LoadRunner Documentation Set

The LoadRunner documentation set consists of one installation guide, a Controller user's guide, and three guides for creating Virtual User scripts.

Installation Guide

For instructions on installing LoadRunner, refer to the *LoadRunner Installation Guide*. The installation guide explains how to install:

- the LoadRunner Controller—on a Windows-based machine
- Virtual User components—for both Windows and UNIX platforms

Controller User's Guide

The LoadRunner documentation pack includes one Controller user's guide:

The *LoadRunner Controller User's Guide (Windows)* describes how to create and run LoadRunner scenarios using the LoadRunner Controller in a Windows environment. The Vusers can run on UNIX and Windows-based platforms. The Controller user's guide presents an overview of the LoadRunner testing process.

Guides for Creating Vuser Scripts

The LoadRunner documentation pack has three guides that describe how to create Vuser scripts:

- The *Creating Vuser Scripts* guide describes how to create all types of Vuser scripts. When necessary, supplement this document with the *LoadRunner Online Function Reference* and one or more of the following guides:
- The *WinRunner User's Guide* describes in detail how to use WinRunner to create GUI Vuser scripts. The resulting Vuser scripts run on Windows platforms. The *TSL Online Reference* should be used in conjunction with this document.
- The *Creating GUI Virtual User Scripts (UNIX)* guide describes how to create GUI Vuser scripts using VXRunner, the enhanced version of XRunner.

The resulting Vuser scripts run on UNIX platforms. The *TSL Online Reference* should be used in conjunction with this document.

For information on	Look here...
Installing LoadRunner	*LoadRunner Installation Guide*
The LoadRunner testing process	*LoadRunner Controller User's Guide (Windows)*
Creating Vuser scripts	*Creating Vuser Scripts* guide
Creating and running scenarios, and analyzing results using a:	
Windows-based Controller	*LoadRunner Controller User's Guide (Windows)*

Typographical Conventions

This book uses the following typographical conventions:

1, 2, 3	Bold numbers indicate steps in a procedure.
➤	Bullets indicate options and features.
>	The greater than sign separates menu levels (for example, **File** > **Open**).
Stone Sans	The **Stone Sans** font indicates names of interface elements in a procedure (for example, "Click the **Run** button.").
Italics	*Italic* text indicates names (for example, names of variables or books).
Helvetica	The Helvetica font is used for examples and strings that are to be typed in literally.
<>	Angle brackets enclose a part of a URL address that needs to be typed in.
[ ]	Square brackets enclose optional parameters.
...	In a line of syntax, an ellipsis indicates that more items of the same format may be included.

Chapter •

Part I

Understanding GUI Vusers

1

Introduction

LoadRunner emulates an environment in which thousands of users work with a client/server system concurrently. To do this, LoadRunner replaces the human user with a *virtual user* (Vuser). While a machine can accommodate only a single human user, large numbers of virtual users can work on the same machine at the same time. Any remote machine can be used as a host for still more virtual users.

LoadRunner provides the following Virtual User technologies:

- GUI Vusers, to operate X Windows and Microsoft Windows applications.
- DB Vusers, to run C programs that access servers directly using API calls.
- RTE Vusers, to operate remote terminal emulator applications (UNIX only).

These Vuser types can be used alone or in combination in order to create effective load testing scenarios.

This guide describes how to develop GUI Virtual User scripts for X Windows applications. For information about developing load testing scenarios, refer to your *LoadRunner Controller User's Guide*.

Working with GUI Virtual Users

GUI Vuser technology is specially designed to test client/server systems using graphical user interface (GUI) applications. GUI Vusers emulate the actions of human users. Like a human user, a GUI Vuser submits input to, and receives output from, your applications. Many Vusers interact with the system concurrently, generating load on the server. This enables you to measure the performance of your server under the load of many users and to test the interaction of the server with your software.

The actions of each Vuser are described in a *Virtual User script*. For instance, to test a bank server that services many automatic teller machines (ATMs), you could create a Virtual User script that:

- opens the ATM application
- enters an account number
- enters the amount of cash to be withdrawn
- withdraws cash from the account
- checks the balance of the account
- closes the ATM application

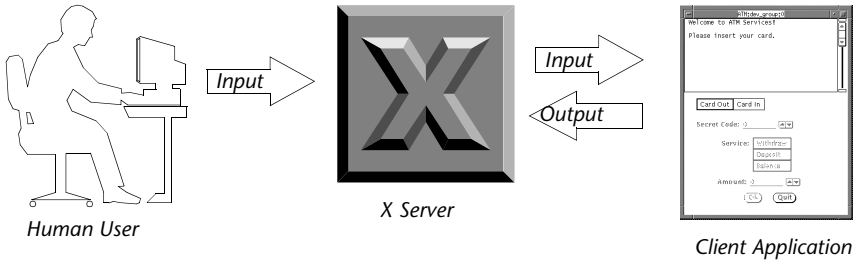
A Virtual User script includes statements that measure system performance during a load-testing session. For example, you can measure how long it takes to check the balance of a bank account. Following a scenario run, you can view performance analysis and other data in reports and graphs.

You can monitor and manage all the virtual users from the LoadRunner Controller. You can run, pause, or view Vusers, and monitor scenario status. Following a scenario run, you can view performance analysis and other data in reports and graphs.

GUI Virtual User Technology

A Vuser emulates the complete UNIX/X environment of a real user. The actual environment for a human user would consist of:

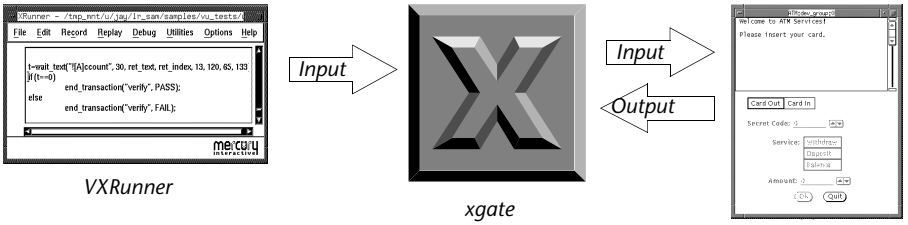
- an X Server
- a client application
- a window manager (optional)



The UNIX environment of a human user.

The virtual user environment consists of:

- VXRunner, an enhanced XRunner, which operates the client application
- a "virtual X Server," which emulates an X server
- the client application
- a window manager (optional)



A LoadRunner GUI Vuser

In the virtual user environment, the person using the client application is replaced by VXRunner, which runs a Virtual User script. VXRunner has no user interface; it is controlled remotely from LoadRunner.

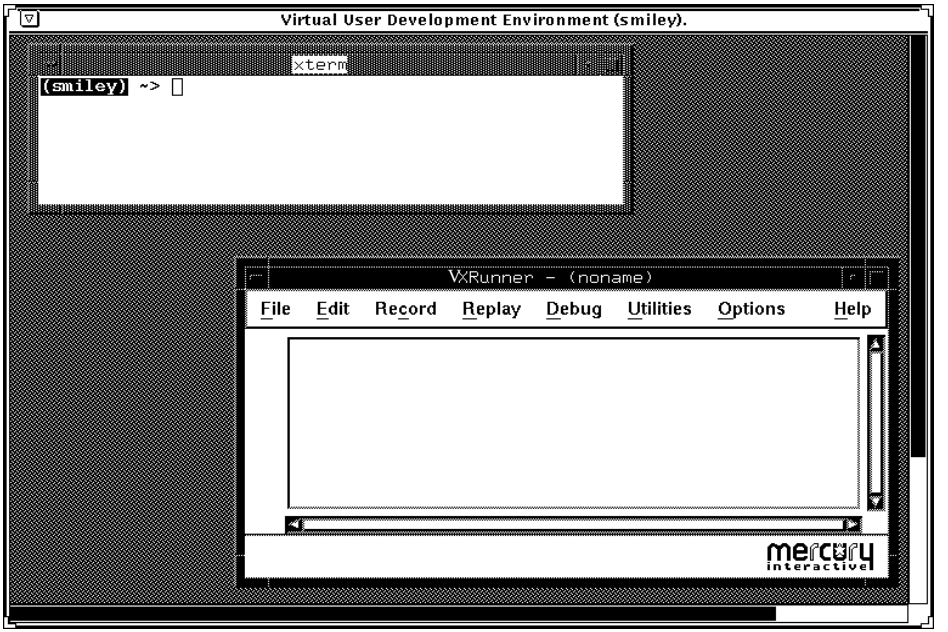
The Virtual X Server is the server on which the client application is activated. The Virtual X Server is an optimized X server that offers a

background mode of operation. In this mode, the Virtual X server receives input from VXRunner only, and none of its X clients appear on the display.

Mercury Interactive's X server technology enables you to run several virtual users simultaneously on a single machine—independent from, and without disturbing one another—while leaving your current display, keyboard and mouse free for regular work. Any time you wish to view a virtual user, LoadRunner can display it on your local host, regardless of the machine on which the Vuser is actually running.

Creating Virtual User Scripts

You create GUI Virtual User scripts in the Virtual User Development Environment (VUDE). This completely independent environment runs in a separate window on your display, and provides all of the benefits offered by Mercury Interactive's Virtual X Server. In this way you can develop Virtual User scripts in the same environment in which you will run them.

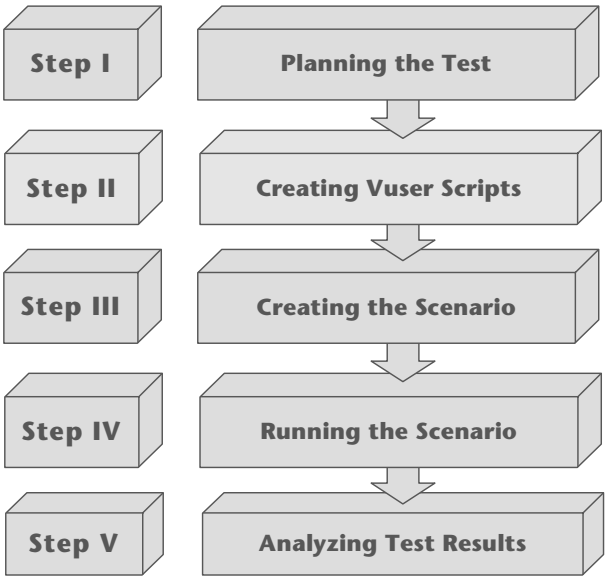


The Virtual User Development Environment contains VXRunner, an enhanced version of XRunner, Mercury Interactive's single-user testing tool. In addition to all of the basic test development tools such as recording, programming, and debugging, VXRunner also supports functions designed especially for multi-user testing.

Virtual User scripts are written in TSL—Mercury Interactive's Test Script Language. TSL is a C-like programming language that is high-level and easy to use. It combines the power and flexibility of a conventional programming language with functions designed specifically for testing. For additional information about TSL, see Chapter 15, "Introducing TSL."

The LoadRunner Testing Process

The following illustration shows the LoadRunner testing process. This guide describes Step II—creating the Vuser scripts. For details on Step I and Steps III–V, refer to your *LoadRunner Controller User's Guide*.



Getting Started with GUI Virtual Users

The following procedure outlines how to develop a GUI Virtual User script and integrate it into a scenario:

1 Open the Virtual User Development Environment.

The Virtual User Development Environment contains a command tool (XTerm) and VXRunner. For more information about the Virtual User Development Environment, see Chapter 2, "Virtual User Development Environment (VUDE)."

2 Invoke your X Windows applications.

Using the command tool inside the Virtual User Development Environment, invoke the applications that the Vuser will use. For more information on invoking your application, see Chapter 2, "Virtual User Development Environment (VUDE)."

3 Create the Virtual User script.

Using VXRunner, create the Virtual User script—using a combination of recording and programming. For more information about creating Virtual User scripts, see Chapter 3, "Recording GUI Virtual User Scripts" and Chapter 15, "Introducing TSL."

4 Insert transactions into the Virtual User script.

Transactions enable LoadRunner to measure system performance during a scenario run. For more information about transactions, see Chapter 12, "Measuring System Performance Using Transactions."

5 Add rendezvous points to the Virtual User script.

Rendezvous points control the execution of Virtual User scripts to emulate intense user load on the server. For more information, see Chapter 13, "Emulating Server Load: Rendezvous Points."

6 Replay the Virtual User script.

You replay the script to validate that it is functional. If necessary, debug the script. For more information, see Chapter 4, "Replaying GUI Virtual User Scripts" and Chapter 9, "Debugging GUI Vuser Scripts."

7 Save the Virtual User script and exit the Virtual User Development Environment.

8 Integrate the Vuser script into a scenario.

You incorporate the script into a LoadRunner scenario by defining the name and path of the Virtual User script that you developed, and the type of GUI Vuser that will run the Virtual User script.

You also set additional attributes, such as the host on which the Vuser runs, the display on which it can be viewed and the window manager that the client application will use.

For more information, refer to your *LoadRunner Controller User's Guide*.

Execute the scenario—Load, Run, and Stop the Vusers.

Use the LoadRunner Controller to manipulate the Vusers.

View reports and graphs.

After running a scenario, view the performance analysis reports and graphs.

2

Virtual User Development Environment (VUDE)

A Vuser script describes the actions of a Vuser. You create GUI Vuser scripts using the Virtual User Development Environment (VUDE).

This chapter describes:

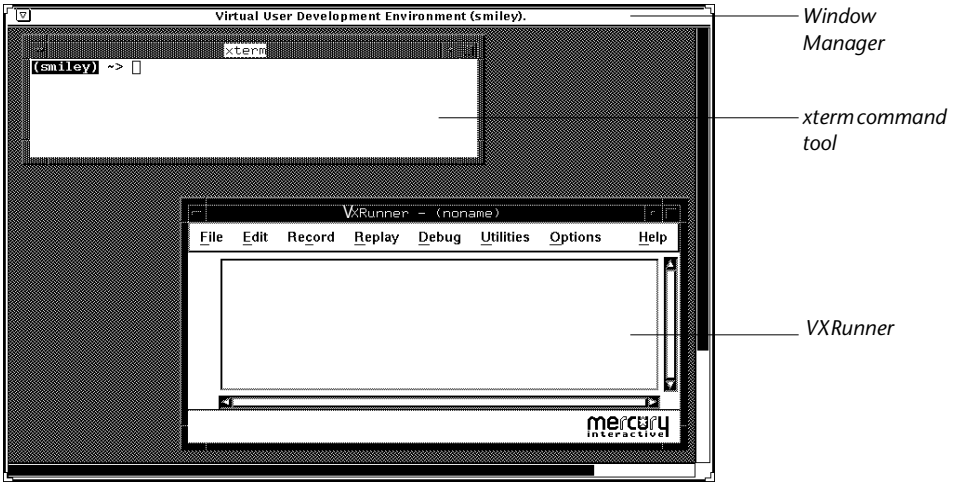
- Creating the Vuser Development Environment (VUDE)
- Closing the VUDE

About the Virtual User Development Environment

You use LoadRunner's Virtual User Development Environment (VUDE) to create GUI Vuser scripts that run on UNIX platforms. The VUDE is a window on your display that runs its own, independent environment, based on Mercury Interactive's Virtual X Server technology.

The VUDE window includes the following components:

- a window manager
- xterm, a command tool to invoke applications
- VXRunner, a version of XRunner modified specifically for use with LoadRunner.



Introducing xterm

The xterm command tool is included in the VUDE so that you can invoke applications for development purposes. If you plan to access a command tool from within a Vuser script, be sure to invoke a separate instance of the command tool. You can do this in either of two ways:

➤ From within the Vuser script using the **system** function. For more information about the TSL **system** function, see the *TSL Online Reference*.

Introducing VXRunner

VXRunner is a complete Vuser script development tool that has been specially adapted for client/server load testing.

Note: VXRunner supports Analog recording only. Unlike XRunner, VXRunner does not support Context Sensitive recording. However, you can take advantage of the full functionality of XRunner in a scenario by designating XRunner as a virtual user in your scenario script. For more information, refer to the *XRunner User's Guide*.

Using VXRunner, you can:

- **Record:** Operate an application. Your actions are recorded and transcribed as TSL statements in a Vuser script. For details, see Chapter 3, "Recording GUI Virtual User Scripts."
- **Program:** Program a Vuser script from scratch or enhance a recorded script. For more information on programming, see Chapter 15, "Introducing TSL."
- **Replay:** Replay the Vuser script in the VUDE to ensure that it executes properly. If necessary, debug the script. For details, see Chapter 4, "Replaying GUI Virtual User Scripts" and Chapter 9, "Debugging GUI Vuser Scripts."

Creating the Vuser Development Environment (VUDE)

To create the VUDE:

1. Start an xgate process. At the UNIX prompt, type:

   ```
   xgate -bypass : &
   ```

   The machine issues a message indicating the listening port. For example:

   ```
   @XGateListens:5975:tiger:1000
   ```

2. Start the window manager, providing a display name and the port. Use the port number issued in the previous message. For example, to start the *mwm* window manager on the host *tiger* on port 1000, type:

   ```
   mwm -display tiger:1000 &
   ```

   A virtual display window opens with the display ID in the title bar.

3 Create an xterm window specifying the host and port. For the above host, type:

```
xterm -display tiger:1000 &
```

A moveable xterm opens inside the virtual display window.

4 Start VXRunner by typing the following in the xterm:

```
vxrun_ui
```

Closing the VUDE

To close the VUDE, choose **File** > **Exit** in the VXRunner window.

Part I

Working with VXRunner

3

Recording GUI Virtual User Scripts

To create a GUI Virtual User script you can use recording, programming, or a combination of both. Usually, you begin by recording a basic script and then enhance the script by programming additional TSL statements.

This chapter describes:

- Recording a GUI Vuser script
- Guidelines for Recording
- Converting Existing XRunner Scripts

For more information about programming, see Chapter 15, "Introducing TSL."

About Recording GUI Virtual User Scripts

When you record a GUI Virtual User script, VXRunner creates a script consisting of TSL statements. These statements describe the sequence of commands and data that VXRunner sends to an application when you replay the script.

VXRunner records keyboard input, mouse clicks, and the precise coordinates traveled by the mouse across the screen. For example, when select the File > Open in an application, VXRunner records the movements of the pointer on the screen. When you replay the script, *VXRunner* returns the mouse pointer to the same coordinates.

Note: VXRunner supports Analog recording only. Unlike XRunner, VXRunner does not support Context Sensitive recording. However, you can take advantage of the full functionality of XRunner in a scenario by designating XRunner as a virtual user in your scenario script. For more information, refer to the *XRunner User's Guide*.

Recording a GUI Vuser script

To record a Vuser script:

1. Open the Virtual User Development Environment (VUDE).
2. Invoke your application using the command tool provided with the VUDE.
3. Select Record > Start Recording, or press the corresponding softkey (F4). VXRunner starts recording the Vuser script.
4. Position your application window. Record its location by pressing the WAIT_WINDOW softkey. (See the last page of this manual for a list of the default softkey definitions.)
5. Using the keyboard and mouse, perform the sequence of operations you want the Vuser to perform on your application.

Note: VXRunner records mouse and keyboard input only from within the VUDE. If you move the mouse out of the window, VXRunner does not record any actions.

---

6. To stop recording, select Record > Stop Recording, or press the STOP softkey. (See the last page of this manual for a list of the default softkey definitions.) VXRunner stops recording the Vuser script.
7. Select File > Save to save the script.

Note: If you do not save a script and you quit the VUDE, all unsaved changes to all open Vuser scripts are discarded.

When you have stopped recording, you can edit your Vuser script and program additional TSL statements. For instance, you can enhance a Vuser script using loops and other control-flow structures. You will also want to program statements that define transactions and mark synchronization points. For more information, see Chapter 12, "Measuring System Performance Using Transactions", and Chapter 13, "Emulating Server Load: Rendezvous Points."

Guidelines for Recording

Consider the following guidelines when recording a GUI Vuser script:

- Before you start recording, close all applications not required for the test.
- You can invoke applications from within the GUI Vuser script by using the **system** function. For more information, see Chapter 7, "Invoking Applications with VXRunner."
- Create your script so that it "cleans up" after itself. When the script is completed, the environment should be as it was at the beginning of the script. For example, if you started with the application window closed, then the script should also close the window—and not minimize it to an icon. This helps ensure accurate replay.
- If the size of your application window can change, resize the window to ensure a consistent size and placement during replay.
- The first time an application window appears on the screen, press the WAIT WINDOW softkey. (See the last page of this manual for a list of the default softkey definitions.) This ensures that during replay, VXRunner moves the window to the correct location and waits for it to be completely redrawn before continuing the replay.
- Avoid typing ahead. For example, when you want to open a window, wait until it is completely redrawn before continuing work.

➤ Avoid holding down the mouse when this results in a repeated action (for example, using the scroll bar to move the screen display). Doing so can initiate a time-sensitive operation that cannot be precisely recreated. Instead, use discrete, multiple clicks to achieve the same results.

Converting Existing XRunner Scripts

In addition to creating Vuser scripts specifically for LoadRunner, you can use existing XRunner scripts. The following limitations apply to XRunner scripts that are replayed by VXRunner:

➤ Context Sensitive functions may not be used.

➤ Vuser scripts must be created using XRunner version 2.0 or later.

➤ All **check_window** statements are treated as **wait_window** statements.

If you designate XRunner as a Vuser, you can run unconverted XRunner scripts as part of a scenario. For more information, see your *LoadRunner Controller User's Guide*.

4

Replaying GUI Virtual User Scripts

Once you have developed a basic GUI Virtual User script, you replay the script to check that it functions as you planned.

This chapter describes:

- Replaying a GUI Vuser script
- Stopping Script Execution
- Pausing Script Execution

About Replaying GUI Virtual User Scripts

Once you have created all or part of a Virtual User script, replay it in order to see that it runs properly, and debug it if necessary.

You can replay a GUI Vuser script in either of two modes: Replay and Verify.

- **Replay** mode is used to execute a Vuser script without performing any verification.
- **Verify** mode is used to compare the current behavior of an application to its behavior during a previous run.

Note: The Verify mode is included in VXRunner solely for reasons of compatibility with XRunner. When developing GUI Vuser scripts for load testing purposes, there is no reason to execute a script in the Verify mode.

Replaying a GUI Vuser script

To replay a GUI Vuser script:

1. Open the script.
2. Move the execution marker to the first line of the script by clicking the left mouse button in the left margin next to the first line of the script.
3. Select Replay > Animate or Replay > Run, or press the corresponding softkeys. (See the last page of this manual for a list of the default softkey definitions.)

   The **Run** command executes the script in the VXRunner window, starting from the line marked by the execution marker. If you do not interrupt the run (by pressing the PAUSE or STOP softkey), execution stops when the script is completed.

   The **Animate** command is the same as the Run command, but the execution marker indicates the line that VXRunner is currently processing. If the script calls another script, the called script is displayed in the VXRunner window. Once the called script is completely processed, the calling script is displayed again.

   To debug a script, you can also replay line by line and define breakpoints. For details, see Chapter 9, "Debugging GUI Vuser Scripts", and Chapter 10, "Using Breakpoints."

Stopping Script Execution

You can stop script execution by pressing the STOP softkey. This terminates execution immediately. (See the last page of this manual for a list of the default softkey definitions.)

When you stop script execution, all values stored for script variables and arrays are lost, as are functions not loaded using the **load** function. These functions must be recompiled. For more information, see Chapter 17, "Creating Compiled Modules."

The values for system variables, however, are retained when a script execution is stopped. Before running the script again, you can restore the

default values of system variables by clicking Default in the Controls dialog box, which you open using the Options menu.

Pausing Script Execution

When you use the PAUSE softkey to stop execution of a GUI Vuser script, the script continues running until all previously interpreted TSL statements are executed. Test variables are not initialized. During a pause, you can access all VXRunner menus. (See the last page of this manual for a list of the default softkey definitions.)

To resume execution of a paused script, select the desired replay command using its softkey. Execution resumes from the point that you paused the script.

5

Synchronizing GUI Vuser Script Execution

Synchronizing your scripts ensures that during execution, VXRunner performs two functions: VXRunner checks the position of a window and relocates it if necessary; VXRunner also delays execution until the window is redrawn or until a specified text string appears.

This chapter describes:

- Synchronizing Script Execution Using wait_window
- Synchronizing Script Execution Using wait_text

About Synchronizing Vuser Script Execution

You use synchronization functions to control the timing of script execution. By inserting synchronization points in your Vuser scripts, you ensure that VXRunner performs operations on your applications at the right time. For instance, assume that your Vuser script opens a terminal window and types in a command at the prompt. A human user would naturally wait for the window to open up, redraw, and come into focus, and for the prompt to appear before typing in the command. Using synchronization functions, you can instruct LoadRunner to wait in order to ensure accurate replay.

Another important use of synchronization functions is in transactions. Transactions measure the amount of time it takes for a Vuser to perform a specific task. Suppose you want to measure the amount of time it takes for the bank to accept a deposit from an automatic teller. The Vuser types in $50 and presses the confirm button. You know that the operation is completed when the message “Done” appears. You use the synchronization functions to control the execution of the transaction in a precise way; LoadRunner

measures only the interval between the click and the appearance of the message.

For more information about defining transactions, see Chapter 12, "Measuring System Performance Using Transactions."

While developing a Vuser script, you can define synchronization points using the **wait_window** and **wait_text** functions.

- The **wait_window** function instructs VXRunner to wait for the appearance of a specified window before continuing script replay. For more information about the **wait_window** function, see "Synchronizing Script Execution Using wait_window" below.
- The **wait_text** function instructs VXRunner to wait for the specified text to appear in a given window before continuing script replay. It provides a method of synchronizing transactions whose beginning or end result is text. For more information, see "Synchronizing Script Execution Using wait_text," on page 29.

Synchronizing Script Execution Using wait_window

The **wait_window** function tells VXRunner to wait for a specific window to appear before continuing script execution.

The syntax of the **wait_window** function is:

wait_window (*time*, *image*, *window*, *width* , *height*, *x*, *y* **);**

- *time* is the interval between the previous input event and the generation of the **wait_window** statement, in seconds. This parameter is added to the *timeout* variable during replay.
- *image* is always an empty string.
- *window* is a string expression indicating the name in the window banner.
- *width, height* are the size of the window, in pixels.
- *x*, *y* are the position of the upper left corner of the active window.

In GUI Vuser scripts, the **wait_window** function waits for a window to appear on the screen and remain stable for the interval defined by the *delay*

system variable. The function also checks the position of the window. If the window does not appear at the coordinates specified in the **wait_window** statement, VXRunner moves the window to the correct position.

Note that VXRunner does not capture or compare bitmap images. The value of the *image* parameter is always a null string (""). Rather, only data related to the window is saved. During replay, VXRunner uses this data to identify and position the window to be redrawn before continuing execution—the contents of the window are not evaluated.

For more information about the **wait_window** function, see the *TSL Online Reference*.

In the following example, a **wait_window** statement is used to delay script execution until the specified window is redrawn. After the window is redrawn, a text string is typed into the window.

```
move_locator_abs(391, 196, 0);
rc = wait_window(4, "","cmdtool -
/usr/local/bin/tcsh",855,802,292,88);
If (rc == 0)
      type ("<t6>ls \-l<kReturn>");
else
      texit;
```

Note: The execution of the **wait_window** function is affected by the current values specified for the following system variables: *timeout, delay, move_windows,* and *raise_windows*. These values can be modified using the Controls dialog box or from within the script using the **setvar** statement. For information on how these variables affect the **wait_window** function, see Chapter 22, "Synchronizing Problematic Windows."

Generating wait_window Statements

You can generate a **wait_window** statement in the following ways:

➤ automatically, by pressing the WAIT_WINDOW softkey.

➤ manually, by typing the statement into your Vuser script.

To generate a wait_window statement automatically:

1. If you are not currently in the Record mode, select Record > Start Recording.
2. Place your mouse pointer anywhere within the desired window.
3. Press the WAIT WINDOW softkey. (See the last page of this manual for a list of the default softkey definitions.) A **wait_window** statement is generated in your script.

   For example, if you place the mouse pointer in the window of the DrawTool drawing application after the zoomed object is displayed and press the WAIT WINDOW softkey, the resulting **wait_window** statement might be:

```
wait_window (35, "", "DrawTool", 800, 600, 100, 120);
```

During Replay

When the script is played back and a **wait_window** statement is interpreted, VXRunner performs the following operations:

➤ Waits for a window to appear that has a DrawTool banner, a width of 800, and a height of 600 pixels. (Note that if you assign a negative value to the *width* and *height* parameters, VXRunner ignores the window size.)

➤ Checks that the window is brought up in the same position as during recording (coordinates 100, 120), and repositions the window if necessary.

Unnamed Windows

If the window you instruct VXRunner to wait for has no banner, the *window* parameter will be an empty string, as follows:

wait_window (*time*, *image*, "", *width*, *height*, *x*, *y*,);

During replay VXRunner waits for an unnamed window image with the specified width and height to appear at the *x*, *y* coordinates.

Note that if the window captured is not recognized by the server, or if an icon is captured, the syntax of the **wait_window** statement will be:

wait_window (*time*, *image*);

Windows with Varying Names

If the window to wait for has a name that varies from run to run, you may edit the *window* parameter so that the window name is a regular expression, rather than a string. For more details see Chapter 20, "Using Regular Expressions."

Synchronizing Script Execution Using wait_text

The **wait_text** function instructs VXRunner to wait for text to appear at a given location before continuing script replay. The function has the following syntax:

wait_text (*pattern*, *timeout*, [*ret_text, ret_index*, *x1*, *y1*, *x2*, *y2*, *ret_bbox*] **);**

- *pattern* is the text VXRunner waits for. This can be a text or null string, or a regular expression.
- *timeout* is the number of seconds that VXRunner waits for the text to appear. By default the *timeout* is equal to the *timeout* system variable.
- *ret_text* is an output variable that stores the actual string that LoadRunner identified as matching the *pattern*.
- *ret_index* is the index of the subexpression that was matched. If *pattern* is a string *ret_index* will equal one when matched. However, if *pattern* is a regular expression it may include a number of *or* operators. In these cases, *ret_index* contains the index of the matched *or* subexpression.
- *x1,y1,x2,y2* are the coordinates of a rectangle that encloses the text to be read. The pairs of coordinates designate the two diagonally opposite corners of the rectangle.
- *ret_bbox* is an optional array that describes the exact location of the text string within the enclosed rectangle. The array also follows the format x1, y1, x2, y2.

Note: Note: When using the **wait_text** function:

The spaces returned by the **wait_text** function are dependent on the application being run. To see the results that a **wait_text** statement will

return, preview the text when you generate a **wait_text** statement. To preview the string that will be captured, press the middle mouse button. The string is displayed directly beneath the selected text.

If the text specified in the *pattern* parameter appears on the display for only a very short time, the VXRunner may not be able to locate the text.

Waiting for Single Strings

In the following example, the **wait_text** statement waits for the appearance of the message "Done" within a certain location on the screen. The *timeout* is set to five seconds.

```
# Wait for the string "Done" to appear.
r = wait_text ("Done", 5, ret_text, ret_index, 0, 0, 500, 500 );
```

You can also generate **wait_text** statements to wait for empty (null) strings. This enables you to instruct the VXRunner to pause script execution until text is erased. For instance, you could program a **wait_text** statement to record the end of a transaction when the "Done" message is erased. To instruct VXRunner to wait for an empty string you set the *pattern* parameter to "".

In the following example, the **wait_text** statement waits for an empty string to appear at a specified location. The *timeout* is set to five seconds.

```
# Wait for text to disappear.
r = wait_text ("", 5, ret_text, ret_index, 0, 0, 500, 500 );
```

Waiting for Multiple Strings

You can instruct VXRunner to wait for one or more strings by using logical operators. Logical operators may be included in the *pattern* parameter if the parameter is a regular expression.

For example, the function call:

```
wait_text ("!OK\| Error", 10, ret_text, ret_index);
```

sets the *ret_index* parameter if either the "OK" or the "Error" string is found. The exclamation point is specific to LoadRunner and is not part of the regular expression. If the OK string is found, the *ret_text* is assigned the string "OK", and *ret_index* is assigned the value 1. If the "Error" string is found, *ret_text* is assigned the string "**Error**", and *ret_index* is assigned the value 2.

The following is a more complex example:

```
pattern = "!\\(\xrunner\|lrunner\) error\)\|OK";
wait_text (pattern, 10, ret_text, ret_index);
```

In this case, the **wait_text** expression will return a value if any of the following strings appear:

```
"xrunner error": (ret_text == "xrunner error", ret_index ==1)
"lrunner error": (ret_text == "lrunner error", ret_index ==1)
"OK" : (ret_text == "OK",          , ret_index ==2)
```

The *or* operator separating the "**xrunner**" and "**lrunner**" is not counted as a subexpression for the *ret_index* value.

The grouping operators "\(" and "\)" are limited to a maximum of 10 pairs in each **wait_text** statement. The *or* operator is not limited.

Generating wait_text Statements

You can generate a **wait_text** statement in either of the following ways:

- automatically, by pressing the WAIT_TEXT softkey
- manually, by typing a **wait_text** statement into your Vuser script.

To generate a wait_text statement automatically:

1. Place the mouse pointer in the Vuser script at the place where you want the **wait_text** statement.
2. Press the WAIT_TEXT softkey. (See the last page of this manual for a list of the default softkey definitions.) The mouse pointer becomes a cross-hairs.
3. Drag the cross-hairs to enclose the text in a rectangle.

To preview the string that will be captured, press the middle mouse button. The string is displayed directly beneath the text. However, if there is not enough space to the right, the string to be captured is displayed in the upper left corner of the screen.

4 Click the right mouse button. VXRunner inserts a **wait_text** statement in your Vuser script.

Waiting for the Re-appearance of a Specified String

The **expect_text** function ensures that **wait_text** statements accurately synchronize transactions. The **wait_text** function monitors all strings in the given rectangle. If the string you defined in the **pattern parameter** is already displayed, **wait_text** will return a result immediately and continue script replay. The transaction you measure will not accurately reflect the time taken to perform this task.

Suppose you create a Vuser script that deposits $50 and then withdraws $50 from an ATM. The text window in the ATM application is not refreshed after the Vuser makes the deposit. When the Vuser selects the withdraw option the message "Done" is still displayed in the ATM window. However, the **wait_text** function sees the message "Done" and instructs LoadRunner to stop measuring the "withdraw" transaction before the action has been performed.

By inserting **expect_text** statements into your Vuser script you can ensure that **wait_text** function waits for the re-appearance of the specified string. The **expect_text** function instructs VXRunner to ignore all the text currently displayed. The syntax of the **expect_text** function is:

```
expect_text ( );
```

Note that inserting an **expect_text** statement before a **wait_text** statement that waits for an empty string is meaningless.

A Sample Synchronized Transaction

In the following example, the deposit transaction is defined to measure how long it takes for a Vuser to deposit fifty dollars using the ATM application. The **expect_text** statement instructs VXRunner to ignore all text currently displayed. The **wait_text** function instructs VXRunner to wait for the

"Done" message to appear. When the message appears, script replay resumes and the duration of the deposit transaction is recorded.

```
# Ignore the text in the ATM window.
expect_text ( );

# Mouse pointer moved to deposit button.
move_locator_abs (10, 10, 0);


# Start measuring deposit operation.
start_transaction ("deposit");

# Click left mouse button on deposit button.
click ("Left");

# Wait for "Done" to appear in the ATM window.
r = wait_text ("Done", 5, ret_text, ret_index, 0, 0, 500, 500, ret_bbox);


# End Deposit transaction.
if (r == 0)
      end_transaction ("deposit", PASS);
else
      end_transaction ("deposit", FAIL);
```

6

Reading Text from the Screen

VXRunner can read text from the graphical user interface (GUI) of an application, and then perform various tasks with the text that is read.

This chapter describes how to develop a GUI Virtual User script that includes:

- Reading Text
- Searching for Text
- Comparing Text

About Text Recognition

Text recognition allows you to:

- read text from the screen, using the **get_text** function.
- search for text on the screen, using the **find_text** function.
- compare two text strings, using the **compare_text** function.

You read text from the screen using the **get_text** function. The **get_text** function returns one line of text from a specified area of the screen, and assigns the text to a variable.

To search for text, you use the **find_text** function. The **find_text** function performs the reverse process of **get_text**. Whereas **get_text** accesses any text found in the designated area, **find_text** looks for a specified string and returns its location on the screen. This location is expressed as a pair of x,y coordinates that delineate a rectangle.

The **compare_text** function compares two strings, ignoring any specified differences. It may be used in conjunction with the **get_text** function, or separately.

Reading Text

You read text from the screen using the **get_text** function. The **get_text** function returns one line of text from a specified area of the screen, and assigns the text to a variable.

Generating get_text Statements

You can generate a **get_text** statement in the following ways:

- automatically, by pressing the GET TEXT softkey.

  (See the last page of this manual for a list of the default softkey definitions.)

- manually, by typing the statement into your Virtual User script.

Manually programming a get_text statement

The **get_text** function can be programmed into the script, using either of the following two syntax formats:

```
variable = get_text(x,y);
```

The *x* and *y* parameters define the coordinates of a single pixel on the screen. The variable is assigned the value of the string closest to this pixel. (The search radius around the specified point is defined by the XR_TEXT_SEARCH_RADIUS parameter. For details, see Appendix A, "VXRunner Configuration Files.")

```
variable = get_text ();
```

When no parameter is specified (the parentheses are empty), the string closest to the position of the mouse pointer is read. (The search radius is defined by the XR_TEXT_SEARCH_RADIUS configuration parameter. For details, see Appendix A, "VXRunner Configuration Files.")

Example Using get_text

The following script segment searches an application for the input name (read from an array). When the name is found, VXRunner reads the contents of the field containing the associated address. The address is read from the application window using the **get_text** function.

Both the input name and address are then printed to an external report file. This search and print operation is repeated until the script fails to find an address for the last input name.

```
i = 0;
do {

    # mouse is brought to Search Address command
    move_locator_track(5);
    mtype ("<kLeft>");
    # name is input from the name array
    type (name[i]);
    # input name is searched for
    type ("<kCR>);
    # acquire contents of address field
    AddressForName = get_text (100,34,150,50);
    printf ("Name : %s, Address : %s \n",
            name[i],
            AddressForName) >> "u/bart/srch_res.rep";
    i++;
} while (SearchResult != "");
close ("u/bart/srch_res.rep");
```

Searching for Text

To search for text, you can use the **find_text** function, which performs the opposite function of **get_text**. Whereas **get_text** accesses any text found in the designated area, **find_text** looks for a specified string and returns its location on the screen. This location is expressed as a pair of x,y coordinates that delineate a rectangle.

The **find_text** function must be programmed in the script, using the following syntax:

find_text (*regular_expression***,** *identifier***,** *area***);**

The parameters included in this statement are:

- *regular expression* specifies a literal, case-sensitive string (in which case it must be enclosed between quotation marks), or the name of a string variable. In the latter case, the value of the string variable can include a regular expression.

  The regular expression should not include blank spaces. The regular expression does not have to begin with a quotation mark.

- *identifier* is the name assigned to the four-element array in which the location of the found string is stored.

- *x* and *y* specify the region of the screen in which the search is to be executed. The area is defined as a pair of coordinates. The values *x1*,*y1*,*x2*,*y2* can specify any two diagonally opposite corners of the region to be searched.

The **find_text** function returns a Boolean value indicating whether the search was successful (1 no, 0 yes). In addition, the function generates the coordinates of the rectangle which bounds the string matching the regular expression. These coordinates are stored in a four-element array specified by the *identifier* parameter in the **find_text** statement.

The elements of the array are numbered 1 to 4. Elements 1 and 2 store the x and y coordinates of the upper left corner of the rectangle; elements 3 and 4 store these coordinates for the lower right corner of this rectangle.

Moving the Pointer to a String

The **move_locator_text** function searches for the specified string in the indicated area of the screen. The position of the string is specified in terms of the rectangle that encloses it. Once the text is located, the mouse pointer is moved to the center of the rectangle. For more information, see the *TSL Online Reference*.

Clicking on a Specified Text String

The **click_on_text** function searches for a specified string in the indicated area of the screen, moves the mouse pointer to the center of the string, and enters a sequence of mouse button clicks. For more information, see the *TSL Online Reference*.

Comparing Text

The **compare_text** function compares two strings, ignoring any specified differences. It may be used in conjunction with the **get_text** function, or separately.

The **compare_text** function has the following syntax:

variable = **compare_text (**str1, str2[,chars1, chars2]**);**

- *str1* and *str2* represent the literal strings or string variables to compare.
- The optional parameters *chars1* and *chars2* represent the literal characters or string variables to be ignored during comparison. Note that *chars1* and *chars2* may specify multiple characters.

The **compare_text** function returns the value 1 when the compared strings are judged to be the same, and 0 when the strings are different.

For example, a portion of your script compares the text string "File" returned by the **get_text** function. Because the lowercase "l" character has the same shape as the uppercase "I", you specify that these two characters be ignored.

```
t = get_text (10, 10, 90, 30);
if (compare_text (t, "File", "l", "I"))
      move_locator_abs (10, 10);
```

7

Invoking Applications with VXRunner

You can run an application from within VXRunner by including its command line in your Virtual User script.

This chapter describes the **system** function that is used to invoke applications.

About Running Applications from within VXRunner

You can run applications from within VXRunner by using the **system** function. The **system** function has the following syntax:

system ("*expression* [&]**");**

The *expression* parameter designates the system command to be executed (including command line options). For example, the script line

```
system ("ls > filelist");
```

causes the contents of the current directory to be written to the file *filelist*.

Note that you can run the invoked application in the background by adding an ampersand (&) character to the **system** statement. This means that after the **system** statement is processed, VXRunner will continue processing the TSL script, even if the application has not yet been invoked. If no ampersand is added, processing of the remainder of the TSL script pauses until the **system** function is completed.

It is recommended to run an application in the background when the application being invoked is interactive.

The **system** statement is interpreted by a Bourne shell, and therefore can include only Bourne shell commands.

Using the System Command to Start an Application

By including the appropriate command line options within a **system** statement, you can specify the exact location at which the application window is displayed. For example, the TSL statement

```
system ("calctool -Wp 300 400&");
```

invokes the calculator application so that the upper left corner of its window is located at screen coordinates 300, 400. By immediately following this script line with the statement

```
wait_window (3,"", "calculator",123, 234, 300, 400)
```

you can instruct VXRunner to wait until the window is completely redrawn before continuing execution.

8

Viewing Execution Reports

Every time you run a GUI Vuser script, VXRunner generates an execution report that details the major events that occurred during the run.

This chapter describes:

- Displaying Execution Reports
- Viewing Reports During Script Execution
- Adding Messages to Reports

About Execution Reports

An execution report contains details about script execution. The exact nature of the information in the report depends on whether the script was run in Replay or in Verify mode.

An execution report includes the following sections:

- A *report header* which details the name of the script; the date of execution; operator name; and miscellaneous comments included in the Test Header.
- A *report summary* with details on the success and duration of execution, and data pertinent to window or file captures performed.
- A *detailed description* of the major events that occurred during the execution run. These can include the start and termination times of the test; windows, or files captured; calls to other scripts; changes to system variables; instances of displayed report messages; and run-time errors.

Execution Report Format

```
To:   Arthur
From: Ford
CC:   Marvin

Date:  Mon June 16 12:03:07 1996

Subject:Test Report

Test name:          /home/qa/calculator/mode tst
Test Results name:  ver 4
Date:               Mon June 16 12:03:07 1996
Operator name:      Charlie
Operator notes:     Third run on June 16

Summary:

Process termination: OK
Total number of checked windows: 1
Process duration time: 00:00:41
Total windows sync time: 3 sec 764 milli sec

Detailed Results Description

#           Event       Result      Name        Test        Lin
Time

1 start-run             ---         ---         mode tst    1
00:00:00
2 Message:The time is Mon Nov 18 09:31:50
3 wait-window           not found   Win 1       mode_tst    6
00:00:17
```

The following information can appear in an execution report:

Report Header Section

To: A free text field.

From: Author of the script (supplied by the system).

CC: A free text field.

Date: Report print date and time.

Test Name: The name of the executed script.

Test Results name: The name assigned to the verification results generated for the current execution run.

Date: The date and time of the script run.

Operator name: Name of the user who ran the script.

Operator notes: A free text field.

Report Summary Section

Process Termination: Indicates whether the script was executed to completion (OK) or prematurely terminated (ABORT).

Total number of checked windows: Number of captured windows (for a Replay run).

Process duration time: The total time (in hr:min:sec) that elapsed from start to finish of the current script run.

Total windows sync time: The total time (in seconds and milliseconds) that was spent on synchronizing window events.

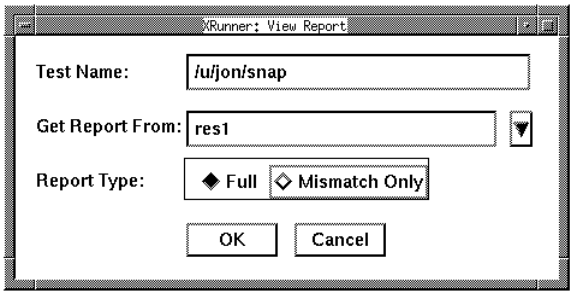
Detailed Description Section

Detailed Results Description: Each major *Event* in the execution of the script is accompanied by its *Result*; the object involved in the event (for example, a called script) is designated by its *Name*; the *Script* and *Line* number in which the event occurred are indicated; *Time* is the amount of time (in hr:min:sec) that elapsed from the start of the run until the occurrence of the event.

Displaying Execution Reports

To display an execution report:

1 Select Utilities > Reports. The View Report dialog box appears.



By default, the "Get Report From" text area displays the directory of the last run, and the "Report Type Full" is selected.

2 Click OK.

The Report is displayed. You can edit the report as desired.

Viewing Reports During Script Execution

During execution of a script, you can view the report currently generated for the script. Simply pause the script and select Utilities > Reports. The name of the current script appears in the *Test Name* field of the Reports dialog box. The displayed report includes only the Detailed Results Description section.

Adding Messages to Reports

You can include your own message in an execution report by inserting a **report_msg** statement in the script:

report_msg (*message***);**

The *message* can be a string, variable, or both. For example, the following **report_msg** statement gets the current value of the *searchpath* system variable, and enters a statement in the execution report containing your message and the current value of *searchpath*:

```
x = getvar ("searchpath");
report_msg ("The current searchpath is" & x);
```

Part II

Debugging GUI Vuser Scripts

9

Debugging GUI Vuser Scripts

VXRunner provides several line-by-line replay commands that enable you to debug your GUI Vuser scripts.

This chapter describes:

- Running a Single Line of a GUI Vuser Script
- Running a Section of a GUI Vuser Script
- Pausing Script Execution

About Debugging GUI Vuser Scripts

VXRunner lets you replay scripts line-by-line in order to isolate and eliminate defects in your scripts. You can use three commands to control replay of statements and functions: Step, Step Into and Step Out. Another controlled replay command is Step to Cursor. This lets you replay segments of your scripts between two specified points.

You can also control script execution by setting breakpoints. A breakpoint pauses a script run at a predetermined point. For more information, see Chapter 10, “Using Breakpoints.”

To help you debug your tests, VXRunner allows you to monitor variables in a script. You define the variables you want to monitor in a Watch List. As the test runs, you can view the values that are assigned to the variables. For more information see Chapter 11, “Monitoring Variables.”

Running a Single Line of a GUI Vuser Script

You can perform controlled execution by selecting the Step, Step Into, or the Step Out commands from the Replay Menu, or by pressing the corresponding softkeys. (See the last page of this manual for a list of the default softkey definitions.)

- The **Step** command executes the current line of the script (the line indicated by the execution marker). When the current line calls another script or user-defined function, the called script or function is executed in its entirety, but is not displayed in the VXRunner window.
- The **Step Into** command, like Step, processes a single line of the current script. However, when the current line of the executed script calls another script or a user-defined function, the called script or function is displayed in the VXRunner window. The individual lines of the called script or function can then be executed using either Step or Step Into.
- The **Step Out** command is used after a called script or function was entered using the Step Into command. Step Out animates to the end of the called script or function and then pauses. Step Out eliminates the need to execute a called script or function line-by-line using the Step command.

Running a Section of a GUI Vuser Script

The Step to Cursor command allows you to perform an animated replay of one section of a script.

To use the Step to Cursor command:

1. Move the *execution marker* to the line of the script where you want execution to begin. Then execute one line by selectiong Replay > Step (or by pressing the STEP softkey [F7]).
2. Move the *insertion point* to the line where you want execution to stop.
3. Select Replay > Step to Cursor. VXRunner executes the script up to the line you marked in step 2.
4. To resume execution, select a command from the Replay menu or press the appropriate softkey. (See the last page of this manual for a list of the default softkey definitions.)

Pausing Script Execution

You can interactively stop the execution of a script by using the Pause command. To pause script execution, select Replay > Pause, or press the PAUSE softkey. To resume execution of a paused script, activate the desired replay command using its softkey. The execution continues from the point that you invoked Pause, or from the position of the execution marker if you moved it while the script was suspended.

10

Using Breakpoints

A breakpoint stops script execution at any point in a script. You can use breakpoints together with the controlled execution commands to identify flaws in your scripts.

This section describes:

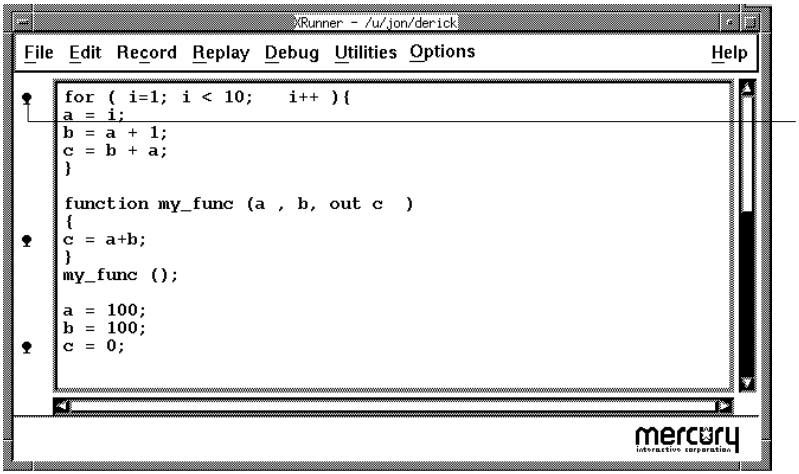
- Setting and Removing Breakpoints
- Modifying Breakpoints
- Deleting a Breakpoint

About Breakpoints

Setting a breakpoint tells VXRunner to stop execution at a specified line or function in a Vuser script. During replay, VXRunner halts before executing the specified line. The script execution can be restarted from that point. Once restarted, it continues to run until it is completed or the next breakpoint is reached. Breakpoints can be set in any script, compiled module, or function.

Breakpoints are created, modified, and deleted using the Breakpoints dialog box. They can also be set using the mouse, the Toggle Breakpoints command, or the Break in Function command.

Breakpoints are indicated by markers that appear in the left margin of the VXRunner window.



You can set a pass counter for each breakpoint to define how many times the breakpoint is passed before execution stops. For example, suppose you create a loop that performs a command fifty times. The pass counter is set by default to zero so VXRunner stops execution each time the loop is performed. If you set the pass counter to 25, execution stops only after the twenty-fifth iteration of the loop.

There are two types of breakpoints: Break at Line and Break in Function.

A **Break at Line** breakpoint is defined by a script name and a line number. The breakpoint marker appears in the left margin of the VXRunner window, next to the designated line. For example, a Break at Line breakpoint could be listed in the Breakpoint dialog box as:

```
my_test[137]:0
```

The breakpoint appears in the script, my_test, at line 137. The number following the colon is the pass counter, here set to zero.

A **Break in Function** breakpoint is defined by the name of a function or a compiled module in the script. The breakpoint marker appears in the left margin of the VXRunner window next to the first line of the function

definition. VXRunner halts execution each time the specified function is called. For example, a Break in Function breakpoint could be listed in the Breakpoints dialog box as:

```
my_func[derick:25]:10
```

The function appears in the script derick. The breakpoint appears in the line containing the function my_func, in this case line 25. The pass counter is set to ten.

Setting and Removing Breakpoints

You can set breakpoints in a variety of ways, as described below.

To set a Break at Line breakpoint using the mouse:

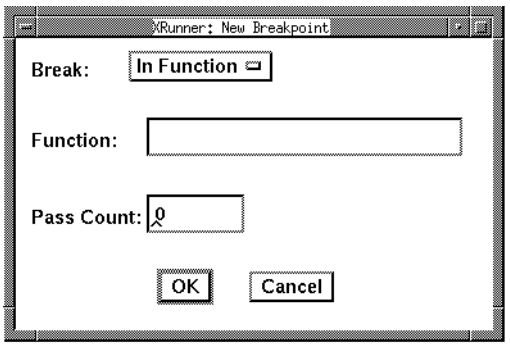
1. Move the mouse pointer to the left margin of the line of the script where you want execution to stop.
2. Click the right mouse button. The breakpoint symbol appears in the left margin of the VXRunner window.
3. Click the right mouse button again to remove the breakpoint.

To set a Break at Line breakpoint using the Toggle Breakpoint command:

1. Move the insertion point to the line of the script where you want execution to stop.
2. Select Debug > Toggle Breakpoint, or press the BREAKPOINT softkey. (See the last page of this manual for a list of the default softkey definitions.) The breakpoint symbol appears in the left margin of the VXRunner window.
3. Select Toggle Breakpoint again to remove the breakpoint.

To set a Break in Function breakpoint Using the Break in Function Command:

1 Select Debug > Break in Function (or press CTRL+B.) The New Breakpoint dialog box opens.



2 Enter a function name in the Function text box. The function must be one already compiled by VXRunner.

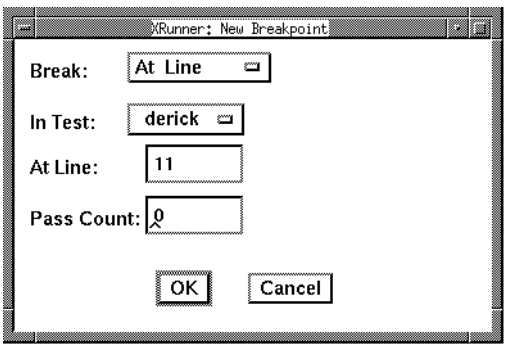
3 Type a value in the Pass Count text box.

4 Close the dialog box by clicking OK. The breakpoint symbol appears in the left margin of the VXRunner window.

To set a Break at Line breakpoint using the Breakpoints dialog box:

1 Select Debug > Breakpoints.The Breakpoints dialog box opens.

2 Click New. The New Breakpoint dialog box opens.



3 The Break box is set to At Line, by default. The script name is set to the current active script, by default. The At Line box is set to the line number of the insertion point. The Pass Counter is set to zero. You can change any of these values. See "Modifying Breakpoints" on page 58.

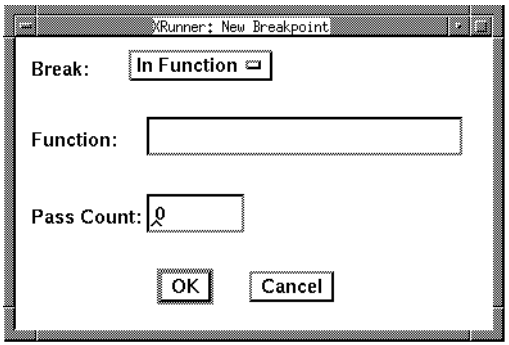
4 Click OK to set the breakpoint and close the New Breakpoint dialog box. The breakpoint appears in the Break at Line list. The breakpoint marker appears in the left margin of the VXRunner window, next to the selected line.

To set a Break in Function breakpoint using the Breakpoints dialog box:

1 Select Debug > Breakpoints. The Breakpoint dialog box opens.

2 Click New. The New Breakpoint dialog box opens.

3 In the Break list, click In Function. The dialog box changes in order to let you type in a function name and a pass count value.



4 Type a function name in the In Function text box. The function must be one already compiled by VXRunner.

5 Type a value in the Pass Count text box.

6 Close the dialog box by clicking OK. The breakpoint appears in the Break in Function list. The breakpoint symbol appears in the left margin of the VXRunner window.

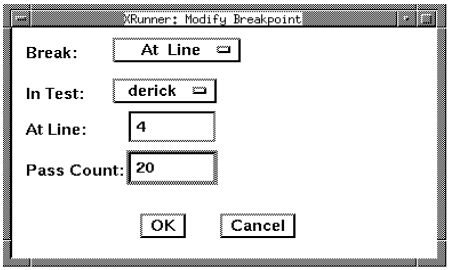
Modifying Breakpoints

You can modify breakpoints by using the Breakpoints dialog box.

To modify a breakpoint:

1 Open the Breakpoints dialog box. Click on a breakpoint in one of the list boxes. The breakpoint is highlighted.

2 Click Modify. The Modify breakpoint dialog box opens.



3 To change the type of Breakpoint, click the Break list and then click a breakpoint type.

4 To select another script, click the In Test list. Click another script in the calls chain.

5 To change the location of the Breakpoint, type a new line number in the At Line Text box.

6 To change the Pass Count, type a new number in the Pass Count text box.

7 Click OK to close the dialog box.

Deleting a Breakpoint

You can delete a single breakpoint or all the breakpoints listed in the Breakpoints dialog box.

To delete a single breakpoint:

1 Open the Breakpoints dialog box.

2 Click a breakpoint in either of the two breakpoint lists. The breakpoint is highlighted.

3 Click Delete. The breakpoint is removed from the list.

4 Click OK to close the Breakpoints dialog box.

To delete all the breakpoints listed in the Breakpoints dialog box:

1 Open the Breakpoints dialog box.

2 Click Delete All. All the breakpoints are deleted from both lists.

3 Click OK to close the dialog box.

11

Monitoring Variables

The Watch List monitors specified variables and expressions during debugging. Use this feature to check the value of variables and to observe how they influence script execution. The Watch List can be used with any variable, array, or TSL expression.

This chapter describes:

- Adding a Variable or Expression to the Watch List
- Adding an Array to the Watch List
- Modifying an Expression in the Watch List
- Assigning a Value to a Variable
- Deleting Expressions and Variables from the Watch List

About Monitoring Variables

The Watch List helps you to debug scripts by monitoring the value of variables, arrays and array elements, and legal TSL expressions. The variables in the Watch List are updated each time VXRunner stops execution (after a Step command, stop on a breakpoint, etc.) You can modify the expressions in the Watch List, and assign new values to variables. In the following script, the Watch List is used to measure and track the value of the variable *a*.

```
for (i = 1; i < 10; i++){
a = i;
b = a + 1;
c = a + b;
}
```

The following expressions and values appear in the Watch List:

```
a:1
a+1:2
b+a:3
```

After the script is run the Watch List shows the following results:

```
a:9
a + 1:10
b + a:19
```

If a test script has several variables with the same name but different scopes, the variable is evaluated according to the current scope of the interpreter. For example, suppose both *test_a* and *test_b* use a static variable *x*, and *test_a* calls *test_b*. If you include the variable *x* in the Watch List, the value of *x* displayed at any time depends on whether VXRunner is interpreting *test_a* or *test_b*.

Selecting a script from the Calls List also changes the context of watch variables and expressions, and updates the Watch List.

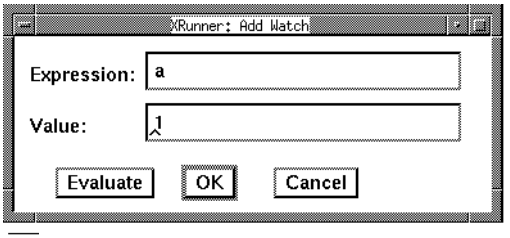
Adding a Variable or Expression to the Watch List

You can add variables to the Watch List by selecting Debug > Add Watch, or by using the Watch List dialog box. Variables can include expressions, arrays, and array elements.

To add a variable to the Watch List using the Add Watch Command:

1. Select Debug > Add Watch (or press Ctrl-W). The Add Watch dialog box opens.
2. Type the variable name in the Expression box. Click Evaluate to see the current value of the variable.

If the variable has not been executed or contains an error, the message "<cannot evaluate>" appears in the Value box.



3 Click OK. The Add Watch dialog box closes and the expression appears in the Watch List.

Note: Do not add expressions to the Watch List that assign or increment the value of variables, because this can affect script execution.

To add a variable to the Watch List using the Watch List dialog box:

1 Select Debug > Watch List to open the Watch List dialog box.

2 Click the Add button to open the Add Watch dialog box.

3 Type the variable name in the Expression box. Click Evaluate to see the current value of the variable.

If the variable was not executed or contains an error, the message "<cannot evaluate>" appears in the Value field.

4 To close the Add Watch dialog box, click OK. The dialog box closes and the expression appears in the Watch List.

Adding an Array to the Watch List

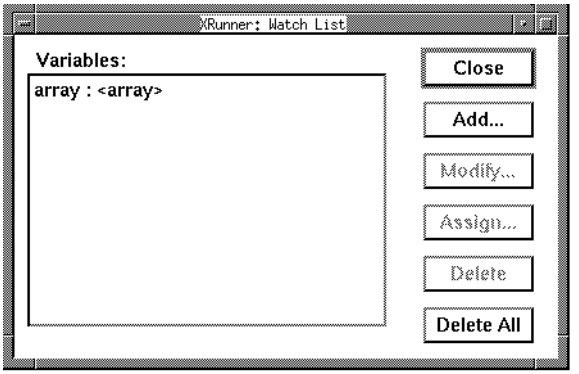
To add an array to the Watch List:

1 Select Debug Add Watch (or press Ctrl-W). The Add Watch dialog box opens.

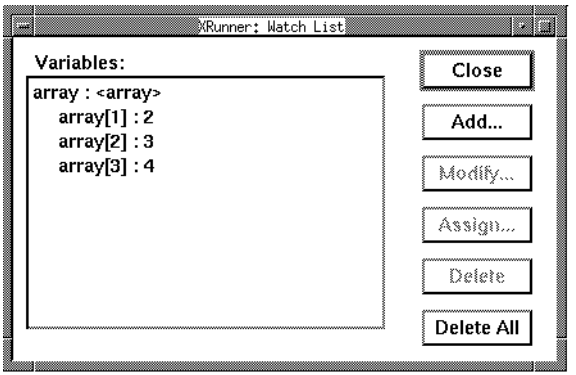
2 Type the array variable in the Expression box. Type the value of the array in the Value box.

3 To close the Add Watch dialog box, click OK.

4 Select Debug > Watch List. The Watch List dialog box opens with the array variable displayed.



5 To view the array elements, double-click the item containing the array variable. Click the array to return to the array variable.



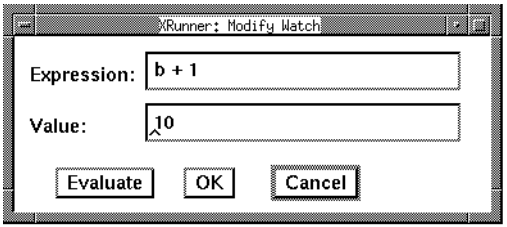
Modifying an Expression in the Watch List

You can modify variables and expressions using the Modify Watch dialog box. For example, you can turn variable *b* into the expression *b + 1*. The

Watch List is automatically updated to produce a new value for the expression.

To modify an expression in the Watch List:

1. Open the Watch List.
2. Click on an expression.
3. Click Modify to open the Modify Watch dialog box.



4. Type changes in the Expression box. Click Evaluate to see the value of the modified expression. The expression is evaluated again and the new value appears in the Value field.
5. Click OK to close the dialog box. The modified value appears in the Watch List.

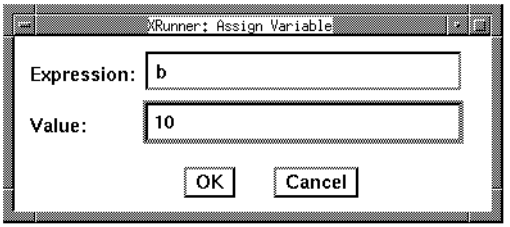
Assigning a Value to a Variable

You can assign new values to variables. For example, you can change the value of variable *b* from 2 to 10. Values can be assigned only to variables or array subscripts, not to TSL expressions.

To assign a value to a variable:

1. Open the Watch List.
2. Click on an expression in the Watch List.

3 Click Assign to open the Assign Variable dialog box.



4 Type the new value in the New Value box.

5 To close the dialog box, click OK. The new value appears in the Add Watch dialog box.

Deleting Expressions and Variables from the Watch List

To delete an expression or variable:

1 Open the Watch List.

2 Click on an expression in the Watch List. The expression is highlighted.

3 Click Delete to remove the expression from the list.

4 Click Close to close the Watch List.

Deleting All Expressions and Variables

To delete all expressions and variables in the Watch List:

1 Open the Watch List.

2 Click Delete All. All expressions are deleted from both lists.

3 Click Close to close the dialog box.

Part III

Using LoadRunner Functions

12

Measuring System Performance Using Transactions

LoadRunner measures server performance by measuring the time taken to perform certain tasks or *transactions*. In the GUI Vuser script, you define the transactions to be measured.

This chapter describes:

- Declaring Transactions
- Marking the Start of a Transaction
- Marking the End of a Transaction

About Measuring System Performance

When you develop a GUI Vuser script you insert *transaction* statements into the script. LoadRunner uses the transactions to measure the time it takes for a Vuser to perform a specific task. This enables you to measure how your system performs under various load conditions.

A transaction may be a single task—such as a deleting a file—or it may include multiple tasks. Within a script, you can mark an unlimited number of transactions for analysis, each with a different name, and starting and ending in different places. Transactions can be nested.

To use the transactions to measure intense server load, you can define *rendezvous* points. For more information, see Chapter 13, "Emulating Server Load: Rendezvous Points."

You can analyze system performance using a variety of graphs and reports. For more information about analyzing system performance, refer to the *LoadRunner Controller User's Guide*.

Declaring Transactions

You must declare all the transactions in a Vuser script in the beginning of the script. To declare a transaction, you use the **declare_transaction** function. The syntax of this functions is:

declare_transaction (*transaction_name* **);**

The *transaction_name* parameter can be any string. It must be a literal string constant, and not a variable or an expression.

In the following example, the **declare_transaction** function is used to declare the transactions "deposit", "withdraw" and "balance":

```
declare_transaction ( "deposit" );
declare_transaction ( "withdraw" );
declare_transaction ( "balance" );
```

Marking the Start of a Transaction

To indicate the start of the transaction, you insert a **start_transaction** statement into your Vuser script—immediately before the action you want to measure. The syntax of this function is:

start_transaction (*transaction_name* [, *when*] **);**

- *transaction_name* can be any transaction that you declared using the **declare_transaction** function.
- *when* determines when LoadRunner begins to measure the transaction, and can be set to NOW or ONINPUT. NOW, the default, causes LoadRunner to begin measuring as soon as the **start_transaction** statement is interpreted. ONINPUT causes LoadRunner to begin measuring the transaction only when the first mouse or keyboard input is submitted to the application.

In the following example, **start_transaction** is used to mark the start of the transaction "deposit". LoadRunner begins to measure the transaction as soon as the **start_transaction** statement is interpreted:

```
start_transaction ( "deposit", NOW );
```

Marking the End of a Transaction

To indicate the end of a transaction, you insert an **end_transaction** statement into your Vuser script—after the action you want to measure. The syntax of this function is:

end_transaction (transaction_name [, status] **);**

- *transaction_name* can be any transaction that you declared using the **declare_transaction** function.
- *status* can be set to PASS or FAIL. This tells LoadRunner if the transaction passed or failed. The default is PASS.

In the following example, **end_transaction** is used to mark the end of the transaction "deposit".

```
end_transaction ( "deposit", PASS );
```

A Sample Transaction

In the following sample Vuser script, the "deposit" transaction is defined to measure how long it takes for a Vuser to deposit $50 at an ATM. LoadRunner starts measuring the transaction as soon as the Vuser clicks the OK button.

The transaction passes if the message "Done" appears in the ATM display. If any other message appears, the transaction fails.

```
# Declare the transaction name.
declare_transaction ( "deposit");

# Move mouse to deposit button.
move_locator_abs (127, 198, 0);

# Click left mouse button.
click ("Left");

# Move to amount field.
move_locator_abs (141, 350,0);

# Type in $50.
type ("50");

# Move to the OK button.
move_locator_abs (135, 378, 0);

# Define a Deposit transaction.
start_transaction ("deposit", ONINPUT);

# Click on the OK button.
click ("Left");

# Wait for "Done" to appear in the ATM window.
rc=wait_text ("Done", 5, ret_text, ret_index, 0, 0, 500, 500, ret_bbox);

# End Deposit transaction.
if (rc == 0)
     end_transaction ("deposit", PASS);
else
     end_transaction ("deposit" , FAIL);
```

13

Emulating Server Load: Rendezvous Points

By inserting rendezvous points into your Vuser scripts you can control the actions of multiple Virtual Users. This allows you to emulate specific and peak load conditions on the server.

This chapter describers:

- Declaring a Rendezvous
- Specifying the Point of Rendezvous in a GUI Vuser Script

About Synchronizing Multiple Vusers

When designing a scenario, you will want to synchronize the actions of two or more virtual users. You can do this by creating inter-user synchronization points in your Vuser scripts. An inter-user synchronization point is called a *rendezvous*. A rendezvous is used to:

- emulate interaction between Vusers (for example, two users access the same account at the same time).
- create a specific or peak load (for example, fifty users try to withdraw cash simultaneously from automatic teller machines).

To create a rendezvous in a scenario:

1. Declare the rendezvous at the start of the Vuser script.
2. Designate the point at which the rendezvous will take place in the Vuser script.

For information on how to synchronize the execution of a Vuser script with the responses from an application, see Chapter 5, "Synchronizing GUI Vuser Script Execution."

Declaring a Rendezvous

You must declare all the rendezvous points in a Vuser script at the beginning of the script. This tells LoadRunner which rendezvous points are included in a script before interpreting the entire script.

To declare a rendezvous, you use the **declare_rendezvous** function. The syntax of this functions is:

declare_rendezvous (*rendezvous_name* **);**

where *rendezvous_name* is the name that you defined in the scenario script for the rendezvous point.

In the following example, the **declare_rendezvous** function is used to declare the rendezvous points "load_10" and "load_20."

```
declare_rendezvous ("load_10");
declare_rendezvous ("load_20");
```

Specifying the Point of Rendezvous in a GUI Vuser Script

You designate the point at which the rendezvous will take place, you insert a **rendezvous** statement into the Vuser script. The function has the following syntax:

rendezvous (*rendezvous_name* **)**;

where *rendezvous_name* is the name of the rendezvous that you declared at the beginning of the script.

A Sample Rendezvous

Suppose that while testing the sample bank application, you want to see what happens when ten Vusers simultaneously deposit, and then withdraw, cash from automatic teller machines. In the scenario script you create a rendezvous named “load_10” and define a Vuser rendezvous list of ten Vusers.

In the Vuser script, you would insert the following in the section where the deposit and withdrawal are performed:

```
# Declare rendezvous
declare_rendezvous ("load_10");

# Define rendezvous points
rendezvous ("load_10");
deposit (50);
rendezvous ("load_10");
withdraw (100);
```

14

Enhancing Scripts using LoadRunner Functions

Once you have developed a GUI Vuser script, you can enhance your script with LoadRunner functions.

This chapter describes:

- Sending Messages from Vuser scripts
- Obtaining Virtual User Information
- Specifying Your Own Data for Analysis

About Enhancing Vuser Scripts with LoadRunner Functions

LoadRunner provides many functions that you can use to enhance your Vuser scripts. For example, you can send messages from Vuser scripts, obtain Virtual User information, and specify your own data for analysis.

For details of all functions specific to LoadRunner GUI Vuser scripts, see Chapter 23, "Function Reference."

Sending Messages from Vuser scripts

You can use the **print and printf** functions in Vuser scripts to write to the Vuser standard output. The information in the Vuser standard output is saved in a file called stdout in the directory scenario/result_dir/group/vuser. You can view this file during scenario execution in the Controller's Output window. For more information on the **print and printf** functions, see the *TSL Online Reference*.

During scenario execution, the LoadRunner Output window displays valuable information about scenario status. In addition to the messages automatically sent by LoadRunner, you can send messages from Vuser scripts to the Output window. Note that only messages relating to scenario execution status should be sent to the LoadRunner output window rather than to standard output.

The following functions enable a Vuser script to send information to the LoadRunner Output window:

- The **error_message** function is used to send an error message.
- The **output_message** function is used to send a special notification that is not an error message.

For more information about each of these functions, see Chapter 23, "Function Reference."

Obtaining Virtual User Information

When you execute a scenario, many Vusers may run the same Vuser script. You may want to know which Vuser is running a particular instance of a script. For instance, consider a Group consisting of 100 Vusers. Every Vuser needs to log in to a remote machine using its own name and password.

LoadRunner provides several functions that enable a Vuser script to obtain information about the Vuser that is running the script. These functions include:

- **lr_whoami:** returns the Vuser, Group, and scenario ID for a Vuser.
- **get_host_name:** returns the name of the host machine for a Vuser.
- **get_master_host_name:** returns the name of the LoadRunner Controller host machine.

For more information about each of these functions, see Chapter 23, "Function Reference."

In the following script segment, the Vuser enters a secret code and then deposits $50 in an automatic teller machine (ATM). The Vuser's secret code

is equal to its ID number plus 100. The **lr_whoami** function is used to determine the ID number of the Vuser currently running the script.

```
# Insert bank card.
move_locator_abs(127, 198, 0);

# Click on the Card_in button.
click ("Left");

# Secret code is Vuser id no + 100. Get Vuser id number.
lr_whoami(id, group);
code = id + 100;

# Type the secret code.
type ( code & "<kReturn>");

# Move mouse to deposit button.
move_locator_abs(127, 198, 0);

# Click the deposit button.
click ("Left");

# Move to amount field.
move_locator_abs(141, 350, 0);

# Type in $50.
type ("50" "<kReturn>");

# Move to the OK button.
move_locator_abs(135, 378, 0);

# Click on the OK button.
click ("Left");
```

Specifying Your Own Data for Analysis

User data points instruct LoadRunner to record the value of specified variables. For example, you could define a user data point to measure CPU utilization over a period of time.

You define a user data point by inserting a **user_data_point** statement into your Vuser script. Every time LoadRunner interprets a **user_data_point**

statement, an event is created in the Vuser event file, and the following information is recorded:

- the name of the data point
- the value of the data point
- the time that the data was recorded

You can use LoadRunner's User Defined report and graph facilities to analyze this data. The syntax of the **user_data_point** function is:

user_data_point (*sample_name*, *value* **);**

- *sample_name* is a string that contains the name of the data point.
- *value* contains the value to be recorded.

In the following example, a user data point checks the CPU every minute, and records the utilization.

```
for (i=0;i<100;i++) {
        cpu_val=cpu_check();
        user_data_point("cpu", cpu_val);
        sleep(60);
}
```

Part IV

Programming with TSL

15

Introducing TSL

GUI Vuser scripts are composed of statements coded in Mercury Interactive's Test Script Language (TSL). These statements are either generated automatically (recorded), or are programmed manually.

This chapter describes:

- Constants
- Variables
- Operators
- Control-Flow Statements
- Built-in Functions
- Comments

About TSL

TSL combines general-purpose programming language features (variables, control-flow statements, arrays, user-defined functions), and built-in functions specifically designed for script creation. Certain words are therefore reserved by TSL and may not be used as variable names. Note that TSL is a case sensitive language.

This chapter provides a brief overview of TSL. For more information, see the *TSL Online Reference*.

Constants

TSL supports two types of constants, *strings* and *numbers*. Strings are enclosed within quotes; numbers are either an integer or floating point type. VXRunner identifies the constant type according to its context.

Variables

Variables are the basic data objects manipulated in a script. As with constants, variables can be either a string or a number.

TSL supports the use of static variables. A static variable is local to the script in which it is declared and does not affect a variable having the same name belonging to another script.

Operators

TSL supports six types of operators:

- arithmetical
- string
- relational
- logical
- conditional
- assignment

Arithmetical Operators

The signs used to represent each of the four binary arithmetical operators are:

+ addition
- subtraction
* multiplication
/ division

In addition, TSL provides modulus and exponentiation operators. Their signs are:

%	modulus
^ *or* **	exponentiation

TSL also supports increment and decrement operators for variables:

++	adds 1 to its operand (incremental)
- -	subtracts 1 from its operand (decremental)

The increment and decrement operators may be placed before the variable (++n), or after the variable (n++). As a result, the variable is incremented or decremented either before or after the value is used.

String Operator

The ampersand (&) character is used by TSL to concatenate adjacent strings. For example, the statement

```
X = "ab" & "cd";
```

assigns the value *abcd* to variable X.

Relational Operators

The relational operators used in TSL are:

>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to
= =	equal to
!=	not equal to

Logical Operators

Logical operators are used to create logical expressions by combining two or more basic expressions. TSL supports the following logical operators:

&&	and
||	or
!	not

When evaluated, a logical expression is assigned the value 1 if true and 0 if false.

Conditional Operator

In TSL, the question mark (?) character is the conditional operator. Conditional expressions have the format:

```
expression1 ? expression2 : expression3
```

First, *expression1* is evaluated, and if it is true, *expression2* is evaluated and becomes the value of the expression. If *expression1* is false (zero or null), then *expression3* is evaluated.

Assignment Operators

The following assignment operators can be used to assign a value to a variable:

Sign	Example	Meaning
=	a = b	assign the value of *b* to *a*
+=	a += b	assign the value of *a* plus *b* to *a*
-=	a -= b	assign the value of *a* minus *b* to *a*
*=	a *= b	assign the value of *a* times *b* to *a*
/=	a /= b	assign the value of *a* divided by *b* to *a*
%=	a %= b	assign the value of *a* modulus *b* to *a*
^= *or* **	a ^ =b	assign the value of *a* to the power of *b*, to *a*

Control-Flow Statements

Control-flow statements determine the sequence and conditions in which script statements are interpreted. The control-flow elements supported by TSL include:

- brackets for creating a compound statement from an enclosed list of simpler statements
- *if ... else,* and *switch* statements for decision making
- *while*, *for*, and *do* statements for looping

The following is a summary of the TSL control-flow statements:

{ *statements* **}**
Compound statement.

if (*expression*) *statement***;**
If expression is true, execute statement.

if (*expression*) *statement1***;else** *statement2***;**
If expression is true, execute statement1; otherwise execute statement2.

while (*expression*) *statement***;**
If expression is true, execute statement; then repeat.

for (*expression1***;** *expression2***;** *expression3***)** *statement***;**
Implement expression1. If expression2 is true, execute the statement and evaluate expression3. Repeat loop until expression2 is false.

do *statement* **while** *expression***;**
Execute statement; while expression is true.

switch (expression) {
case case_expr$_1$: statements
case case_expr$_2$: statements
case case_expr$_n$: statements
[**default**: statements]}
Evaluate the case expressions until one is found to be equal to the expression, and execute statements. If no case is equal to the expression, then execute the optional default statements.

break;
Causes an exit from within a loop or a switch.

continue;
Causes the next cycle of a loop to begin.

Built-in Functions

TSL provides an assortment of built-in functions. These are described in detail in the *TSL Online Reference*.

Comments

A number sign (#) in a line of a TSL script indicates that all text located between this sign and the end of the line is a comment. The VXRunner interpreter does not process comments.

16

Creating User-Defined Functions

You can expand VXRunner's capabilities by creating your own, user-defined TSL functions. Functions can appear in a script or a compiled module.

This chapter describes:

- Function Syntax
- Variable, constant, and array declarations
- Return Statement

About User-Defined Functions

In addition to its built-in functions, TSL allows you to design and implement your own functions. The following main options are available:

- You can create user-defined functions in a script. Once the function is replayed, it can be called from anywhere within a script.
- You can create user-defined functions in a compiled module. Once the module is loaded, the function can be called from any script. For more information, see Chapter 17, "Creating Compiled Modules."

User-defined functions are convenient in situations when you want to perform the same operation in a script several times. Instead of repeating the code over and over, you can write a single function that performs the operation. As a result your scripts are modular, more readable, and easier to debug and maintain.

A function can be called from anywhere in the script. Since it is already compiled, execution time is faster. For instance, suppose you create a script that opens a number of files and checks their contents. Instead of recording

or programming the sequence that opens the file several times, you could write a function and call it each time you want to open a file.

Function Syntax

A user-defined function has the following structure:

[*class*] **function** *name* ([*mode*] *parameter*...)
{
declarations;
statements;
}

Class

The class of a function may be either static or public. A static function is available only to the script or module within which the function was defined.

Once you replay a public function, it is available to all scripts, as long as the script containing the function remains open. This is convenient when you want the function to be accessible from called scripts. However, if you want to create a function that will be available to many scripts, you should place it in a compiled module.

If no class is explicitly declared, the function is assigned the default class, public.

Parameters

Function parameters can be of mode *in*, *out*, or *inout*. For all non-array parameters, the default mode is *in*. The significance of each of these parameter types is as follows:

in: A parameter that is assigned a value from outside the function.

out: A parameter that is assigned a value from inside the function.

inout*:* A parameter that can be assigned a value from outside the function, as well as pass on a value to the outside.

Array parameters are designated by square brackets. For example, the following parameter list would indicate that variable *a* is an array:

```
function my_func (a[], b, c){
 ...
}
```

Array parameters can be either out or inout. If no class is specified, the default inout is assumed.

While variables used within a function must be explicitly declared, this is not the case for parameters.

Declarations

Normally in TSL, declaration is optional. In functions, however, variables, constants, and arrays must all be declared. The declaration can be within the function itself, or anywhere else within the script or module. Additional information about declarations can be found in the *TSL Online Reference*.

Variables

Variable declarations have the following syntax:

class variable [**=** *init_expression*]**;**

The *init_expression* assigned to a declared variable can be any valid expression. If an *init_expression* is not set, the variable is assigned an empty string. The variable *class* can be one of the following:

auto: An auto variable may be declared only within a function. It is limited in scope to the function within which it is defined, and exists only as long as the function is still running. Note that a recursive call of the function creates a new copy of an auto variable.

static: A static variable is limited in scope to the function, script, or module within which it is defined.

public: A public variable may be declared only outside a function. Such a variable is available to all scripts.

extern: An extern variable is defined outside of the function, script, or module in which it appears. An extern declaration cannot initialize the variable.

With the exception of the auto variable, all variables continue to exist until the Abort command is executed. The following table summarizes the scope, lifetime, and availability (where the declaration can appear) of each type of variable:

Declaration	Scope	Lifetime	Availability
auto	local	end of function	within function only
static	local	until abort	function, script, or module
public	global	until abort	script or module only
extern	global	until abort	function, script, or module

Note: In compiled modules, the Abort command initializes static and public variables. For more information about compiled modules, Chapter 17, "Creating Compiled Modules."

Constants

The *const* specifier indicates that the declared value cannot be modified. The syntax of this declaration is:

[*class*] **const** *name* [**=** *expression*]**;**

The *class* of a constant may be either public or static. (If no class is explicitly declared, the constant is assigned the default class public.) Once a constant is defined, it remains in existence until you exit VXRunner.

For example, defining the constant TMP_DIR using the declaration:

```
const TMP_DIR = "/tmp";
```

means that the assigned value /tmp cannot be modified. (This value can only be changed by explicitly making a new constant declaration for TMP_DIR.)

Arrays

The following syntax is used to define the class and the initial expression of an array. Array size need not be defined in TSL.

class array_name **[ ]** [=*init_expression*]

The array *class* may be any of the classes listed under Variable Declarations, above.

An array can be initialized using the C language syntax. For example:

```
public hosts [ ] = {"lithium", "silver", "bronze"};
```

This statement creates an array with the following elements:

```
hosts[0]="lithium"
hosts[1]="silver"
hosts[2]="bronze"
```

Note that like in C, arrays with the class *auto* cannot be initialized.

In addition, an array can be initialized using a string subscript for each element. The string subscript may be any legal TSL expression. Its value is evaluated during compilation.

Statements

Any valid statement used within a TSL script can be used within a function, except for the **treturn** statement.

Return Statement

The **return** statement is used exclusively in functions. The syntax is:

return [*expression*]**;**

This statement halts execution of the called function and passes control back to the calling function or script. It also returns the value of the evaluated expression to the calling function or script. If no expression is assigned to the **return** statement, an empty string is returned.

17

Creating Compiled Modules

Compiled modules enhance your TSL programming capabilities, providing all the advantages of a compiled environment within your interpreted script.

This chapter describes:

- Compiled Module Contents
- Creating a Module
- Loading and Unloading a Compiled Module
- Incremental Compilation

About Compiled Modules

A compiled module contains user-defined functions that you want to call frequently from within many different scripts. When you load the module, the functions are compiled and remain in memory. You can call them directly from within any script.

For instance, you might want to create a module which includes functions that:

- initialize an application: position it, and resize the window.
- compare the size of two files.
- handle error messages that your application displays in a popup window.

Using compiled modules, you can improve the organization and performance of your scripts. Compiled modules are libraries of frequently-used functions. Your scripts can use functions from libraries instead of

creating new ones each time. Since you debug compiled modules before using them, your scripts will require less error-checking. In addition, calling a function that is already compiled is significantly faster than interpreting a function in a script.

You can compile a module in one of two ways. You can use the TSL **load** function, or you can replay the function interactively using any of the VXRunner replay commands. If you need to debug a module or make changes, you can use the Step command to perform incremental compilation. You only need to replay the part of the module that was changed in order to update the entire module.

Compiled Module Contents

A compiled module is similar to any regular script you create in TSL: it can be opened, edited, and saved. You indicate that a script is a compiled module by clicking Compiled Module in the Test Header dialog box (see "Creating a Module," in this chapter).

In terms of its content, a compiled module is different from an ordinary script in that it has no database: it cannot include screen captures or any other analog input such as mouse tracking. Remember that the purpose of a compiled module is to store those functions that you use most in a module so that they can be quickly and conveniently accessed from other scripts.

Unlike in a regular script, all data objects (variables, constants, arrays) must be declared before use. The structure of a compiled module is similar to a C program file, in that it may contain the following elements:

- function definitions and declarations for variables, constants and arrays (For more information, Chapter 16, "Creating User-Defined Functions.")
- prototypes of external functions (For more information, see the *TSL Online Reference*.)
- **load** statements to other modules (see "Loading and Unloading a Compiled Module" on page 97)

Note that when user-defined functions appear in compiled modules:

- A public function is available to all modules and scripts, while a static function is available only to the module within which it was defined.
- The loaded module remains resident in memory even when execution is aborted. However, all variables defined within the module (whether static or public) are initialized.

Creating a Module

To create a compiled module:

1. Open a new script.
2. Write the module script.
3. Open the Header dialog box (select File > Header), and click Compiled Module. Click OK.
4. Select File > Save.
5. Save your modules in a location that is readily available to all your scripts. When a module is loaded, VXRunner locates it according to the Search Path.
6. Compile the module by choosing one of the Replay commands or using the **load** function.

Loading and Unloading a Compiled Module

In order to access the functions in a compiled module you need to load the module. You can load it from within any script; all scripts can then access the function until you quit VXRunner or unload the module.

If you create a module that contains frequently-used functions (such as the ones described at the beginning of this chapter), you can load the module from your initialization script. For more information, see Chapter 19, “Creating Initialization Scripts.”

You may load a module either as a *system* module or as a *user* module. A system module is generally a closed module that is “invisible” to the user. It is not animated when it is loaded and it is not stopped by a pause

command. A system module is not unloaded when you execute an **unload()** statement with no parameters (global unload).

A user module is the opposite of a system module in these respects. Generally, a user module is one that is still being developed. In such a module you might want to make changes and incrementally compile them.

load

The **load** function has the following syntax:

load (*module_name***,** [**,**1|0] [**,**1|0]**);**

The *module_name* is the name of an existing compiled module.

Two additional, optional parameters indicate the type of module. The first parameter indicates whether the module is a system module or a user module. 1 indicates a system module. 0 indicates a user module. (Default=0)

The second optional parameter indicates whether the module will appear in the Switch To menu. 1 indicates that the module will not appear in the menu. 0 indicates that the module will appear in the menu. (Default=0)

When the **load** function is executed the first time, the module is compiled and stored in memory. This module is now ready for use by any script and need not be interpreted again.

A loaded module remains resident in memory even when execution is aborted. However, all variables defined within the module (whether static or public) are initialized.

unload

The **unload** function removes a loaded module or selected functions from memory, and has the following syntax:

unload (*module*|*test* **[,***function_name* **]);**

For example, the following statement removes all functions loaded within the script named my_test.

```
unload ("my_test");
```

An **unload** statement with empty parentheses removes all modules loaded within all scripts during the current session, except for system modules.

reload

If you make changes to a module, you can reload it. The **reload** function removes a loaded module from memory, and reloads it (combining the functions of **unload** and **load**).

The syntax of the **reload** function is:

reload (module_name, [,1|0] [,1|0}**);**

The *module_name* is the name of an existing compiled module.

Two additional optional parameters indicate the type of module. The first parameter indicates whether the module is a system module or a user module. 1 indicates a system module. 0 indicates a user module. (Default=0)

The second optional parameter indicates whether the module will appear in the "Switch to" menu. 1 indicates that the module will not appear in the menu. 0 indicates that the module will appear in the menu. (Default=0)

Note: Do not load a module more than once. To recompile a module, use **unload** followed by **load**, or the **reload** function.

If you try to load a module that has already been loaded, VXRunner does not load it again. Instead, it initializes variables and increments a *load counter*. If a module has been loaded several times, then the **unload** statement does not unload the module, but rather decrements the counter. For instance, suppose that script A loads a module *math_functions*, and then calls script B. Script B also loads *math_functions*, and then unloads it at the end of the script. VXRunner does not unload the function; it decrements the load counter. When execution returns to script A, *math_functions* is still loaded.

Incremental Compilation

You can also compile a module by replaying it. This is especially useful when you are developing or modifying a module. If a module has already been loaded, and you modify or add just a few lines, you can replay those statements step by step. The compiled version of the module is automatically updated. Note that if you make a change within a function, you must replay the entire function.

To compile a module by replaying it:

1. If the module is not already open, open it.
2. To load an entire module, move the execution marker to the first line of the script, and select Animate or Run.
3. To incrementally compile part of a module, replay the necessary statements using the Step command.
4. Save the module if required, and close it.

Compiled Module Example

The following module contains two simple functions that you can call from any script. The modules receive a pair of numbers and returns the number with the maximum and minimum value.

```
# return maximum of two values
function max (x,y){
if (x>=y)
      return x;
else
      return y;
}
```

```
# return minimum of two values
function min (x,y){
if (x<y)
      return x;
else
      return y;
}
```

18

Calling Scripts

The GUI Vuser scripts you create with VXRunner can call, or be called by, any other GUI Vuser script. This is done by means of the **call** statement. Using this statement, a script can be invoked from within another script, and parameter values can be passed to the called script.

This chapter discusses:

- Using the Call Statement
- Returning to the Calling Script
- Setting the Search Path
- Defining Parameters

Using the Call Statement

A script is invoked from within another script by means of a **call** statement. A **call** statement has the following syntax:

call *script_name* **(**[*parameter*$_1$, *parameter*$_2$, ...*parameter*$_n$]**);**

Parameters are optional. However, when one script calls another, the **call** statement should designate a value for each parameter defined for the called script. If no parameters were defined for the called script, the **call** statement must include an empty set of parentheses.

Any called script must be stored in a directory specified in the search path, or else must include a full pathname within quotation marks.

While replaying a called script, you can pause execution and view the current call chain. To do so, select Debug > Calls.

Returning to the Calling Script

The **treturn** and **texit** statements are used to stop execution.

- The **treturn** statement stops the current script, and returns control to the calling script.
- The **texit** statement stops execution entirely.

Both functions provide a return value for the called script. If **treturn** or **texit** is not used, or if no value is specified, then the return value of the **call** statement is 0.

The **treturn** statement terminates execution of the called script and returns control to the calling script. The syntax is:

treturn [(*expression*)]**;**

The optional *expression* is the value returned to the **call** statement used to invoke the script. For example:

```
# script a
if (call script b() == "success")
    report_msg("script b succeeded");
```

```
# script b
if
check_window(3,"Win_2","Calc",
20,35,35,45);
    treturn("success");
else
    treturn("failure");
```

Here, if the screen comparison in script_b is successful, then the string “success” is returned to the calling script, script_a. If there is a mismatch, then script_b returns the string “failure” to script_a.

Setting the Search Path

The search path determines the directories searched for a called script. To set the search path, select Options > Search Path. The directories are searched in the order of their appearance in the Search Path dialog box.

To add a directory:

1. Type the directory name in the Pathname box. Alternatively, click a directory in the list, click Copy to copy it to the Pathname box, and edit as needed.
2. Click a directory in the Directories list.
3. Click either Add after or Add before to indicate where to place the directory.

To change a directory:

1. Click a directory in the Directories list.
2. Click Copy to copy the directory to the Pathname text box.
3. Make the desired changes.
4. Click Change.

To delete a directory:

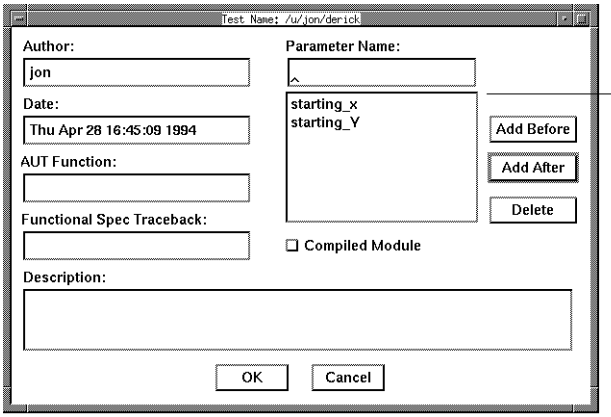
1. Click a directory in the Directories list.
2. Click Delete.

Defining Parameters

A parameter is a variable that is assigned a value from outside the script in which it is defined. You can define one or more parameters for a script; any calling script must then supply values for these parameters.

For example, you might define two parameters (*starting_x* and *starting_y*) for a script. The intended function of these parameters is to assign a value to the initial mouse position when the script is called. Subsequently, two values supplied by a calling script would supply the *x* and *y* coordinates of the mouse pointer.

Parameters are defined for a script using the Test Header dialog box. The parameter list displays the names of all parameters defined for the script.



The Test Header dialog box includes the following fields and buttons:

Parameter Name: A text box where you can type in the name of new or changed parameters.

Author: The name of the script developer.

Date: The day, date and time of script creation (system-supplied).

AUT Function: A text box that you can use to specify the name of the AUT function.

Functional Spec Traceback: A text box that you can use to reference the relevant section in the AUT functional specifications.

Description: A text box that you can use to describe the current script in detail.

Add After/Add Before: Adds the parameter in the text box immediately before or after the highlighted parameter in the list.

Delete button: Deletes the highlighted parameter in the Parameter list.

To define a new parameter:

1. Type the name of the parameter in the Parameter Name box.
2. Click one of the parameters in the list and then click either Add After or Add Before.
3. Note that the order in which parameters are listed determines which value is assigned to a parameter by the calling script, since parameter values are assigned sequentially.
4. Click OK to complete the operation and close the dialog box.

To delete a parameter from the parameter list:

1. Click the name of the parameter to delete.
2. Click Delete.
3. Click OK to complete the operation and close the dialog box.

Parameter Scope

The parameter defined in the called script is known as a *formal* parameter. Test parameters can be constants, variables, expressions, array elements, or complete arrays.

Parameters that are expressions, variables, or array elements are evaluated and then passed to the called script by value. This means that a copy is passed to the called script. This copy is local; if its value is changed in the

called script, the original value in the calling script is not affected. For example:

```
# script 1 (calling_script)
i = 5;
call script 2(i);
print(i);  # prints "5"
```

```
# script 2 (called script), with
# formal parameter x
x = 8;
print (x);  # prints "8"
```

In the calling script (script_1), the variable *i* is assigned the value 5. This value is passed on to the called script (script_2) as the value for the formal parameter *x*. Note that when a new value (8) is assigned to *x* in script_2, this change does not affect the value of *i* in script_1.

Complete arrays are passed by reference. This means that, unlike array elements, variables, or expressions, they are not copied. Any change made to the array in the called script influences the corresponding array in the calling script. For example,

```
# script q
a[1] = 17;
call script r(a);
print(a[1]); # prints "104"
```

```
# script r, with parameter x
x[1] = 104;
```

In the calling script (script_q), element 1 of array *a* is assigned the value 17. Array *a* is then passed to the called script (script_r), which has a formal parameter *x*. In script_r, the first element of array *x* is assigned the value 104. Unlike the previous example, this change to the parameter in the called script does affect the value of the parameter in the calling script, because the parameter is an array.

Once again, with the exception of arrays, formal test parameters are local to the script for which they are defined. Changes made to them do not affect

variables of the same name outside of the script, and their values are lost when the script is completed. For example:

```
# script y
c = 12;
call script z(c);
print(c); # prints "12"
```

```
# script z, with parameter c
c = 42;
```

The value of variable *c* in script_z is changed. However, since this variable is a formal parameter of script_z, it is local. Therefore, the value of variable *c* in script_y is not affected.

All undeclared variables that are not on the formal parameter list of a called script are global and may be accessed and altered from another called, or calling script. If a parameter list is defined for a script, and that script is not called but is run directly, then the parameters function as global variables for the run. For more information about variables, refer to the *TSL Online Reference*.

The script segments below summarize the difference between local and global variables. Note that script_a is not called but is run directly.

```
# script a, with parameter k
i = 1;
j = 2;
k = 3;
call script b(i);
print(j & k & l);
# prints ’2 5 6’
```

```
# script b, with parameter j
j = 4;
k = 5;
l = 6;
print (i & j & k);
# prints ’1 4 5’
```

Part V

Advanced VXRunner Features

19

Creating Initialization Scripts

Using initialization scripts, you can ensure that each time VXRunner is invoked, it is configured correctly for your scripts. This saves time and guarantees uniform conditions.

This chapter describes how to use initialization scripts.

About Initialization Scripts

You can use initialization scripts to customize VXRunner to your requirements. The types of data inserted into an initialization script can include **load** statements that load compiled modules containing user-defined functions.

By creating an initialization script, each time you invoke VXRunner all needed functions are compiled. You can create an initialization script for a group of users, or you can create initialization scripts for each individual user.

Types of Initialization Scripts

The different types of initialization scripts are as follows:

- The first level initialization script is named *tslinit*. This is a system-wide initialization script provided with VXRunner which resides under $M_ROOT/dat. This script contains internal information needed by VXRunner, and should not be changed (any modifications will be erased when you install the next VXRunner).

➤ The second level initialization script is optional. You define the environment variable XR_TSL_INIT so it points to the full path of this script. This script can be used to customize VXRunner for a group of users using the same application. For example:

```
setenv XR_TSL_INIT /qa/share/qinit
```

➤ The third level initialization script (also optional) is the *tslinit* script stored in your home directory. This script can be used to tailor VXRunner to your individual needs.

Each time VXRunner is started, the three scripts are searched for and are run sequentially before the VXRunner window is displayed.

20

Using Regular Expressions

You can use regular expressions in many different ways to increase the flexibility and adaptability of your scripts.

This chapter describes:

- Using regular expressions
- Regular Expression Syntax

About Regular Expressions

Regular expressions can be used in several different ways:

- For synchronization **wait_window** TSL statement: to indicate the window name.
- For text recognition with the **find_text** function: to indicate the string to locate.

VXRunner regular expressions include options similar to some of those offered by the UNIX grep command. For additional information, see the UNIX manpages for ed(1).

Regular Expression Syntax

All regular expressions must begin with an exclamation point (!). Any character that is not one of the special characters described below is searched for literally. When a special character is preceded by a backslash, VXRunner searches for the literal character.

The following options can be used to create regular expressions:

Matching Any Single Character

A period (.) instructs VXRunner to search for any single character. For example,

```
welcome.
```

matches welcomes, welcomed, or welcome followed by a space or any other single character. A series of periods indicates a range of unspecified characters.

Matching Any Single Character within a Range

In order to match a single character within a range, you can use square brackets ([ ]). For example, to search for a date which is either 1968 or 1969, write:

```
196[89]
```

You can use a hyphen (-) to indicate an actual range. For instance, to match any year in the 1960s, write:

```
196[0-9]
```

Brackets can be used in a physical description to specify the label of a static text object that may vary.

A hyphen does not signify a range if it appears as the first or last character within brackets, or after a caret (^).

A caret (^) instructs VXRunner to match any character except for the ones specified in the string. For example:

```
[^A-Za-z]
```

matches any non-alphabetic character. The caret has this special meaning only when it appears first within the brackets.

Note that within brackets, the characters ".", "*", "[" and "\" are literal. If the right bracket is the first character in the range, it is also literal. For example:

[]g-m]

matches the right bracket, and g through m.

Matching One or More Specific Characters

An asterisk (*) instructs VXRunner to match zero or more occurrences of the preceding character. For example:

Q*

causes VXRunner to match Q, QQ, QQQ, etc. In the following **wait_window** statement, the regular expression causes VXRunner to find any text editor window.

```
wait_window (3, "", "!Text Editor.*", 560, 560, 30, 30);
```

Because the asterisk follows a period, VXRunner locates any combination of characters. You could also use a combination of brackets and an asterisk to limit the window banner name to a combination of non-numeric characters only:

```
wait_window (3, "", "!clock-\[ [a-zA-Z]*\]", 560, 560, 30, 30);
```

21

Setting System Variables

VXRunner provides several ways to view and set the system variables that affect script replay.

This chapter describes:

- Setting System Variables from within the Script
- The Controls Dialog Box
- The Test Environment Dialog Box
- System Variables

About System Variables

VXRunner system variables affect various aspects of script replay.

Each system variable has a default value, listed in the system configuration file. System variables can be set before or during script execution. You can set system variables in two ways:

- through the Controls dialog box
- with the **setvar** function

Setting System Variables from within the Script

The **getvar** and **setvar** built-in functions allow you to read and assign values of system variables from within a script. Using these functions, you can locally modify parameter values during execution as required. For example, you might want to conditionally set different replay speeds for certain

sections of the script, depending on the value returned to some user-defined variable. Note that some variables are read-only. The list under **setvar** indicates the variables that can be set.

getvar

The **getvar** function is used to retrieve the current value of a system variable. The syntax of this statement is:

user_variable = **getvar (***system_variable***);**

In this function, *system_variable* may specify any one of the following:

```
beep              kbd_delay          synchronize
click_delay       key_editing        speed
curr_dir          line_no            sync_mode
dblclk_time       mismatch_break     sync_time
delay             move_windows       sysmode
exp               raise_windows      testname
fast_replay       result             timeout
focus_delay       searchpath
```

For example:

```
nowspeed = getvar ("speed");
```

assigns the current value of the replay speed to the user-defined variable, *nowspeed*.

setvar

The **setvar** function is used to set the current value of a system variable from within the script. This function has the syntax:

setvar (*system_variable*, *value***);**

In this function, *system_variable* may specify any one of the following:

```
beep              focus_delay        searchpath
click_delay       kbd_delay          synchronize
curr_dir          key_editing        speed
dblclk_time       mismatch_break     sync_mode
delay             move_windows       sync_time
exp               raise_windows      timeout
fast_replay       result
```

For example:

```
setvar ("mismatch_break", "off");
```

Sets the break on mismatch system variable to off. A variable retains a value until it is reassigned from within the script, or from the Controls dialog box.

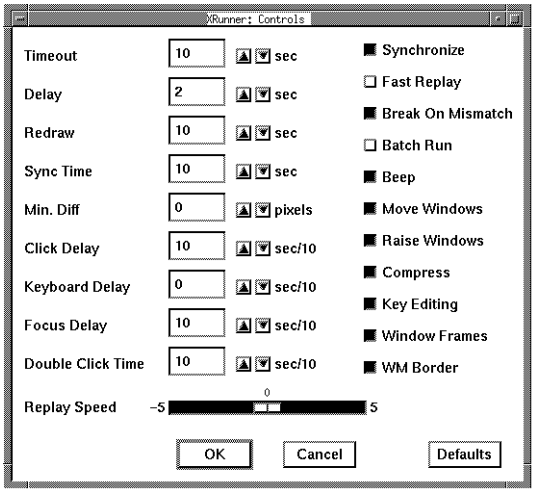
You can use a combination of **getvar** and **setvar** statements to control script execution. For example, in the following script fragment, VXRunner checks the image win_2. The **getvar** and **setvar** functions are used to control the value of the *timeout* and *delay* system variables. The **getvar** statement is used to retrieve the values of *timeout* and *delay*, and **setvar** is used to assign values to these variables. After the window is checked, **setvar** is used to return *timeout* and *delay* to their original values.

```
t = getvar ("timeout");
d = getvar ("delay");

setvar ("timeout", 30);
setvar ("delay", 3);

wait_window (2,"","calculator",261,269,93,42);

setvar ("timeout", t);
setvar ("delay", d);
```

The Controls Dialog Box

The Controls dialog box allows you to set values for system variables. You can set values before script execution. Note that certain system variables cannot be set from the Controls dialog box.

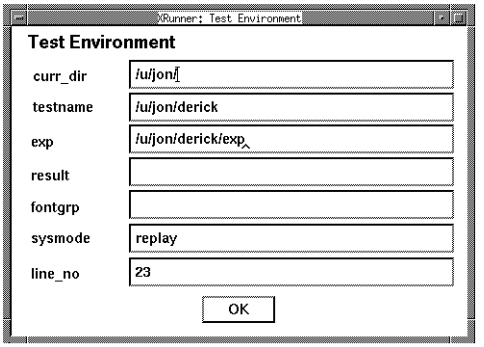
The Controls dialog box appears when you select Options > Controls. Changes made in the dialog box are implemented after you click the OK button.



You can return all system variables to their default values by clicking Defaults and then clicking OK. The default values are determined by the configuration parameters in the system configuration file.

The Test Environment Dialog Box

The Test Environment dialog box contains a list of read-only fields that provide general information about the current script. To view the System Variables dialog box, select Options > Test Environment.



System Variables

The following section describes VXRunner's system variables in detail. The default value for each system variable is listed in Appendix A, "VXRunner Configuration Files."

beep
Causes VXRunner to issue a beep each time a window is checked. The default value is set by the XR_BEEP configuration parameter.

click_delay
Determines the amount of time that VXRunner waits after interpreting a single click of a mouse button. During fast replay, using a longer *click_delay* ensures that two consecutive single clicks are not misinterpreted as a double-click. Note: This does not apply to double-clicks. If a double-click is recorded, it replays as a double-click regardless of the value of the *click_delay*. The *click_delay* is expressed in tenths of a second. The *click_delay* default value is set by the XR_CLICK_DELAY parameter.

curr_dir
Indicates the current working directory for the test. There is no default value for this variable.

dblclk_time
Defines the maximum interval (in tenths of a second) that can constitute a pause between consecutive clicks of a double-click. It is advised to make the *dblck_time* consistent with your system default. The *dblck_time* system variable is listed as double click time on the Controls dialog box. The *dblck_time* default value is set by the XR_DBLCLK_TIME parameter.

delay
During replay when VXRunner reads a **check_window** statement, it captures the window only when it determines that the window is stable. For example, when *delay* is two seconds, and *timeout* is ten seconds, VXRunner checks the AUT window every two seconds until two consecutive checks produce the same results, or until ten seconds have elapsed. The *delay* default value is set by the XR_RETRY_DELAY parameter. Setting the *delay* variable to 0 disables all image checking.

exp
The full pathname of the expected results directory associated with the current execution of the test. There is no default value for this parameter.

fast_replay
Replays tests at the fastest speed at which the AUT is capable of receiving input. Selecting *fast_replay* disables the speed bar in the Controls dialog box. The *fast_replay* default value is set by the XR_FAST_REPLAY parameter.

focus_delay
Determines the interval (in tenths of second) that VXRunner waits from the time the mouse is moved to a new window until input is entered during replay. This variable is particularly important during fast replay. *Focus_delay* ensures that VXRunner does not send keystrokes to the new window before this window is ready to receive them. The *focus_delay* default value is set by the XR_FOCUS_DELAY parameter.

key_editing
Generates more concise type statements so that they represent only the net result of pressing and releasing input keys. This makes your test script easier to read. Whenever the exact order of keystrokes is important for your test, you should disable *key_editing*.

For example, typing the letter "A" with *key_editing* off, produces the following statement:

```
type ("<kShift>-a-<kShift>+a+");
```

With *key_editing* on, the statement is:

```
type ("A");
```

For more information on *key_editing*, see the **type** function in the *TSL Online Reference*. The default value is set by the configuration parameter XR_KEY_EDITING.

kbd_delay
Defines the delay (in tenths of a second) that VXRunner waits after replaying keyboard input. The *keyboard_delay* system variable is listed as KEYBOARD DELAY on the Controls dialog box. The *keyboard_delay* default value is set by the XR_KBD_DELAY parameter.

line_no
Displays the current line of the execution marker in the test script. There is no default value for this variable.

move_windows
Causes VXRunner to automatically return a window to the location specified in a **wait_window** statement.

If you deactivate *move_windows*, you must take measures to ensure that during replay, each AUT window opens at the same screen location as when the test was recorded. The default value is set by the XR_MOVE_WINDOW parameter.

raise_windows
Brings the window to be checked to the front of the screen display during replay. The default value is set by the XR_RAISE_WINDOW parameter.

result
The full pathname of the results directory during a verification run. There is no default value for this variable.

searchpath
The path(s) which VXRunner searches for called tests. The *searchpath* value is set by the XR_SEARCH_PATH parameter.

speed
The speed at which a test script is replayed. At 0, the test is played back using the time parameters recorded in the script. If the speed is set to +5, the test is played back approximately five times faster than the speed at which it was recorded. Setting the speed to -5 causes the test to be executed at approximately one fifth the specified speed.

To set execution speed, either drag the square marker on the speed bar in the Controls dialog box or type the desired value in the Replay Speed field. The values displayed on the Speed Bar can be configured using the XR_SPEED_RANGE parameter.

synchronized
Causes VXRunner to monitor X Server messages and send input to the AUT only when it is ready to receive it. This ensures that even if VXRunner's replay speed is modified, or the AUT responds more slowly during replay than during recording, input will not be lost.

By default, test replay is synchronized (the box is checked). When *synchronized* is not checked, VXRunner ignores synchronization information and replays the test at the speed at which it was recorded. The default value for *synchronize* is set by the XR_SYNCHRONIZED parameter.

sync_time
VXRunner synchronizes test execution by monitoring communication between the AUT and the X Server. When an event is not received, VXRunner waits up to the predefined *sync_time* (in seconds) and then continues execution. The *sync_time* default value is set by the XR_SYNC_TIME parameter.

sysmode
The current mode: either Replay or Verify.

testname
The full pathname of the current test. There is no default value for this parameter.

timeout
The maximum time that VXRunner waits for the execution of **wait_window** statements during replay, before proceeding to the next

statement. The maximum time is calculated by adding the *time* parameter of the statement to the *timeout* variable.

For example, in the statement:

```
wait_window (2, "", "calculator", 120, 240, 85, 150);
```

if the *timeout* variable is 10 seconds, this operation takes a maximum of 12 (2+10) seconds. The default value for the *timeout* variable is set by the XR_TIMEOUT parameter in the system configuration file.

22

Synchronizing Problematic Windows

Some windows that are redrawn slowly require the adjusting of one or more system variables in order to ensure reliable script replay.

This chapter describes:

- How System Variables Affect wait_window Functions
- Adjusting the Timeout Interval
- Setting the Delay

About Synchronizing Problematic Windows

Some "problematic" windows— such as large windows that are redrawn slowly—require the adjusting of one or more system variables in order to ensure reliable execution of **wait_window** statements.

The execution of **wait_window** functions is affected by the values you set for the following system variables:

- timeout
- delay
- move_windows
- raise_windows

The values of system variables can be modified either using the Controls dialog box or from within a script by using the **setvar** statement. For a detailed description of each of these variables and how they can be set, see Chapter 21, "Setting System Variables."

The following examples demonstrate how system variables affect the execution of the **wait_window** function. For each example, assume that the following values are assigned to the VXRunner configuration parameters:

Configuration Parameter	Value
TIMEOUT	10 (*seconds*)
DELAY	2 (*seconds*)
MOVE_WINDOWS	"on"
RAISE_WINDOWS	"on"

How System Variables Affect wait_window Functions

The following example illustrates how various system variables affect the execution of a **wait_window** function.

```
wait_window(7, "", "calctool", 400, 300, 120, 180);
```

When this statement is executed, VXRunner first sets a time limit of 17 seconds for the **wait_window** operation. (This is the value of the *time* parameter plus the *timeout* system variable.) VXRunner then waits for the appearance of a window named *calctool* having a width of 400 and a height of 300 pixels.

Suppose that the calctool window takes 3 seconds to come up on the screen. When it appears, VXRunner repositions the window so that its upper left corner is located at screen coordinate 120, 180. VXRunner now monitors the calctool window to determine if it is stable. If no input is sent to the window for the delay period (2 seconds), then the window is stable and VXRunner moves on to the next line of the script. If there have been changes to the calctool window, VXRunner waits a further 2 seconds before rechecking the window. VXRunner repeats this process until either the window is stable or the time limit (17 seconds) is reached.

Adjusting the Timeout Interval

Drawing applications such as DrawTool allow you to issue a "zoom" command. When you give this command, the program starts a complex and lengthy calculation and the enlarged image is gradually displayed. The content of the window changes continuously until the zoom operation is completed.

In your script, you may want execution to wait until the final zoomed image is displayed before entering any further input to the application. To do this, when recording the script, move the pointer into the window after the zoomed image is displayed and press the WAIT WINDOW softkey. (See the last page of this manual for a list of the default softkey definitions.) A statement like the following is generated in the TSL script:

```
wait_window (56, "", "DrawTool", 800, 600, 100, 100);
```

When the script is played back, this statement fails if the value recorded for the *time* parameter (56 seconds) is too small to allow the lengthy calculation to be completed and the desired image to be displayed.

To ensure that the screen has sufficient time to come up, increase the specified time to 100 seconds. Note that this does not waste time, since VXRunner continues to the next statement as soon as the window is stable.

Setting the Delay

Suppose that you want to wait for a window of DrawTool to be redrawn.You can use the WAIT REDRAW softkey to produce a statement such as the following:

```
wait_window (150, "", "DrawTool", 900, 700, 125, 116);
```

During script replay, you may discover that VXRunner samples the window twice and then moves on to the next line before the window is redrawn. This occurs if the intervals between consecutive samplings are too short. For example, if the redraw starts after 20 seconds and VXRunner samples the

window every two seconds, then after the first two samplings VXRunner concludes that the window is stable and has been completely redrawn.

VXRunner allows you to adapt the script to the behavior of a specific window by controlling the *delay* system variable. For a window that is redrawn slowly, you can use the **getvar** and **setvar** functions from within the script to temporarily increase the delay. For example:

```
old_delay = getvar("delay");

# sample window every 30 seconds
setvar("delay", 30);
wait_window(150,"", "DrawTool",900,700,125,116);

# revert to previous value
setvar("delay", old_delay);
```

In the above example, VXRunner samples the window every thirty seconds. This is enough time for a change to appear in the window. VXRunner therefore concludes that the window is redrawn if no input is sent to the DrawTool window for a 30 second period.

Part VI

GUI Vuser Script Programming Reference

23

Function Reference

The following pages provide an alphabetical reference of all TSL functions specific to LoadRunner GUI Vuser scripts. These functions are defined in the *vxrlib* module, which is loaded automatically by the *tslinit* startup script. For a full list of TSL functions, see the *TSL Online Reference*.

The name of each function, along with a brief description, appears at the top of the page. The following information is also provided for each function:

- complete syntax
- parameter definitions
- details on how the function works
- an example of the function
- return value

Return Values

All LoadRunner functions return one of the following return values. These constants are predefined in the system initialization script.

Code	Name	Description
0	E_OK	Operation successful.
-10001	E_GENERAL_ERROR	A general error. For example, memory allocation has failed.
-10002	E_NOT_FOUND	Object was not found.

Code	Name	Description
-10003	E_NOT_UNIQUE	More than one object has the same name.
-10007	E_FILE_OPEN	Could not open file.
-10008	E_WRITE_ERROR	Could not perform write operation.
-10014	E_OPERATION_ABORTED	The operation was aborted.
-10016	E_TIMEOUT	Timeout was reached before operation could be performed.
-10017	E_COMM	Operation did not succeed due to communication problems.
-10201	E_SG_NF	The specified Group could not be found.
-10202	E_VU_NF	The specified Vuser could not be found.
-10203	E_ATTR	The set attribute is not legal.
-10204	E_VU_LOST	The system has lost the Vuser.
-10205	E_HOST_NF	The given host could not be found.
-10206	E_SHADOWS_NF	The given shadow server could not be found.
-10207	E_PATH_NF	The executable file could not be located.
-10208	E_PARTIAL_ERROR	The operation partially succeeded.
-10209	E_SC_NF	The scenario object could not be found.
-10210	E_NAME_IL	The name is illegal.
-10211	E_PARAMETER_IL	The specified parameter is illegal.
-10212	E_OP_IL	The operation is illegal. For example, deleting a running Vuser or running a Vuser which is not loaded.

Code	Name	Description
-10213	E_RANGE	The provided value is out of range.
-10214	E_FAIL	The operation failed.
-10215	E_APP_NOT_FOUND	The application object was not found.
-10216	E_APP_EXIST	The application object already exists.
-10217	E_NO_LICENSE	The appropriate license was not found.
-10218	E_REND_NF	No such rendezvous was defined in the current scenario.
-10219	E_REND_NOT_MEM	The Vuser was not defined as a participant in the designated rendezvous.
-10220	E_REND_INVALID	The specified rendezvous is currently in the invalid state (valid = off).

declare_rendezvous

declares a rendezvous.

declare_rendezvous (*rendezvous_name* **);**

rendezvous_name A string that is the name of a rendezvous created in the scenario script. The *rendezvous_name* must be a string constant and not a variable or an expression.

Each rendezvous in the Vuser script must be declared at the beginning of the Vuser script.

Example

In the following example, the rendezvous “load_10” is declared:

```
declare_rendezvous ( "load_10");
```

See Also

rendezvous

declare_transaction

declares a transaction.

declare_transaction (*transaction_name* **);**

transaction_name A string that is the name of a transaction created in the scenario script. The *transaction_name* must be a string constant and not a variable or an expression.

Each transaction in the Vuser script must be declared at the beginning of the Vuser script.

Example

In the following example, two transactions are declared:

```
declare_transaction ( "deposit");
declare_transaction ( "withdraw");
```

See Also

start_transaction, end_transaction

end_transaction

marks the end of a transaction for performance analysis.

end_transaction (*transaction_name,* **[** *status* **]);**

transaction_name	A string expression that names the transaction. The string cannot contain any spaces.
status	Optional parameter that tells LoadRunner to measure the transaction if it either passed or failed. Set this parameter to PASS, (0) or to FAIL (any non-zero value). The default value is PASS.

To indicate a transaction to analyze, use the start_transaction and end_transaction functions. These are inserted immediately before and after the transaction, and enable LoadRunner to measure the time it takes for the transaction to be performed.

Transactions can be nested, but each **end_transaction** statement must be associated with a **start_transaction** statement or it will be interpreted as an illegal command. Remember that for the transaction time to be meaningful, you should include a synchronization function before the end of the transaction.

The transaction time is written once the **end_transaction** statement is interpreted. If you do not end the transaction, then no duration time will be recorded. The next time a **start_transaction** statement with the same transaction name is interpreted, the timing restarts at 0.

Note that each transaction in the Vuser script must be declared at the beginning of the Vuser script. You declare transactions using the **declare_transaction** function.

Example

In the following example, the deposit "transaction" measures the time it takes for a Vuser to deposit fifty dollars at an ATM. Once the deposit is completed and returns a value to the variable *status,* the transaction is completed.

```
#Declare the transaction name.
declare_transaction ( "deposit");

# Move mouse to Deposit button.
move_locator_abs (127, 198, 0);

# Click left mouse button.
click ("Left");

# Move to amount field.
move_locator_abs (141, 350,0);

# Type in $50.
type ("50");

# Move to the OK button.
move_locator_abs (135, 378, 0);

# Define a Deposit transaction.
start_transaction ("deposit", ONINPUT);

# Click on the OK button.
click ("Left");

# Wait for “Done” to appear in the ATM window.
wait_text ("Done", 5, ret_text, ret_index, 0, 0, 500, 500, ret_bbox);

# End Deposit transaction.
if (rc == 0)
    end_transaction ("deposit", PASS);
else
    end_transaction ("deposit" , FAIL);
```

Return Values

This function returns 0 if the operation is successful or a non-zero value if the operation fails. For more information, see the Return Values table on page 135.

See Also

start_transaction, declare_transaction

error_message

sends an error message to the Output window.

error_message (*message* **);**

message Any string.

This function enables a Vuser script to send an error message to the LoadRunner Controller. The message is displayed in the Output window during execution, and will be stored with the Virtual User test results. The message appears in the Execution report if the errors option is selected.

Example

In the following example, the Vuser script sends an error message if a specific value is not met.

```
if (ret_code < 0){
    lr_whoami(vu_num, grp_name);
    mess = sprintf("Vuser %s from Group %s can't read the file: %s",vu_num,
grp_name,
    file_name);
    error_message(mess);
    texit(1);
}
```

Return Values

This function returns 0 if the operation is successful or a non-zero value if the operation fails. For more information, see the Return Values table on page 135.

See Also

output_message

expect_text

ignores all the text currently displayed.

expect_text ( );

The **expect_text** function is used in conjunction with the **wait_text** function to synchronize script replay. The **expect_text** function instructs VXRunner to ignore all the text currently displayed in the active window and wait for the string defined in the **wait_text** statement to appear.

Example

In the following example, the deposit transaction is defined to measure how long it takes for a Vuser to deposit fifty dollars using an ATM application. The **expect_text** statement instructs VXRunner to ignore all strings currently displayed in the ATM window. The **wait_text** function instructs VXRunner to wait for the "Done" message to appear. When the message appears, script replay is resumed and the time taken to perform the deposit transaction is recorded.

```
# Ignore the text in the ATM window.
expect_text ( );

# Mouse pointer moved to deposit button.
move_locator_abs (10, 10, 0 );

# Start measuring deposit operation.
start_transaction ("deposit", ONINPUT);

# Click left mouse button on deposit button.
click ("left");

# Wait for the string "Done" to appear.

# Wait for "Done" to appear in the ATM window.
rc = wait_text ("Done", 5, ret_text, ret_index, 0, 0, 500, 500, ret_bbox);

# End Deposit transaction.
if (rc == 0)
    end_transaction ("deposit", PASS);
else
    end_transaction ("deposit", FAIL);
```

Return Values

This function returns 0 if the operation is successful or a non-zero value if the operation fails. For more information, see the Return Values table on page 135.

See Also

wait_text, start_transaction, end_transaction

get_host_name

returns the name of the host that is replaying the current Vuser script.

get_host_name ( );

This function returns the name of the host that is running the current script. Any Virtual User or scenario script can use this function to determine the name of its host.

Example

In the following example the **my_host_name** statement gets the host name and displays it in the Output window and in the execution report.

```
my_host_name = get_host_name();
pause("my local host name is:" & my_host_name);
```

Return Values

This function returns the host name if the operation is successful or null if the operation fails.

See Also

lr_whoami, get_master_host_name

get_master_host_name

returns the name of the LoadRunner host machine.

get_master_host_name ( );

This function enables a Vuser script to determine the host machine of the LoadRunner Controller.

Example

In the following example, the Vuser script reads the Vuser host and the LoadRunner Controller host names. The **print** function sends this information to the standard output file.

```
my_host_name = get_host_name()
master_hostname = get_master_host_name();
print("my local host name is: " & my_host_name);
print("The LoadRunner Controller is running on host: " & master_hostname);
```

Return Values

This function returns the host name if the operation is successful or null if the operation fails.

See Also

get_host_name

lr_whoami

returns information about the Vuser currently executing the script.

lr_whoami (*vuser, group, scenario_id* **);**

vuser	The output variable that stores the id of the Vuser.
group	The output variable that stores the name of the Group.
scenario_id	The output variable that stores the internal id of the scenario.

This function returns information about the Vuser currently executing a Vuser script. In the Virtual User Development Environment, the function will always return the value 0 for *vuser*, "dev_group" for *group*, and the actual scenario id. If you are not interested in one of the values, place a NULL parameter in its place.

Example

In the following example, the Vuser script reads the Vuser information and prints this information to standard output before calling a login function. The pause statement displays this information in a popup window.

```
lr_whoami (vuser, group, NULL);
print ("Virtual User:"& vuser & "Group:"& group &“ starting login...");
```

Return Values

This function returns 0 if the operation is successful or a non-zero value if the operation fails. For more information, see the Return Values table on page 135.

See Also

get_host_name, get_master_host_name

output_message

sends a message to the Output window.

output_message (*message* **);**

message Any string.

This function enables a Vuser script to send a message to LoadRunner. The message will be displayed in the Output window during execution, and will be stored with the Vuser test results. The message will appear in the Execution report, if the errors option is selected.

Example

In the following example, the Vuser script sends a message if a specific value is met.

```
if (ret_code ==0){
    lr_whoami(vu_num, grp_name);
    mess = sprintf("Vuser %s from Group %s is OK: %s",vu_num, grp_name);
    output_message(mess);
    texit(1);
}
```

Return Values

This function returns 0 if the operation is successful or a non-zero value if the operation fails. For more information, see the Return Values table on page 135.

See Also

lr_whoami, error_message

rendezvous

sets a rendezvous point in a Vuser script.

rendezvous (*rendezvous_name* **);**

rendezvous_name A string that is the name of a rendezvous created in the scenario script.

This statement indicates a rendezvous point in a Vuser script. When this statement is interpreted, the Vuser script will stop and the Vuser will wait for permission from LoadRunner to continue.

Note that in the Virtual User development environment, **rendezvous** will not have any effect. It is not possible to test a rendezvous in the Virtual User Development Environment since the environment contains only a single Vuser, which is not part of a scenario. For more information, refer to the *LoadRunner Controller User's Guide*.

Example

In the following example, a rendezvous begins immediately before a deposit transaction.

```
# Start monitoring ATM window for strings.
expect_text ( );

# Set the rendezvous point.
rendezvous ("multi_deposit");

# Define a Deposit transaction.
start_transaction ("deposit");

.
.
.

# End transaction.
end_transaction ("deposit")
```

Return Values

This function returns 0 if the operation is successful, or one of the following error codes if it fails:

Code	Name	Description
0	E_OK	Operation successful.
-10016	E_TIMEOUT	Timeout was reached before operation could be performed.
-10218	E_REND_NF	No such rendezvous was defined in the current scenario.
-10219	E_REND_NOT_MEM	The Vuser was not defined as a participant in the designated rendezvous.
-10220	E_REND_INVALID	The specified rendezvous is currently in the invalid state (valid = off).

See Also

declare_rendezvous

start_transaction

marks the beginning of a transaction for performance analysis.

start_transaction (*transaction_name* **[** *, when* **]);**

transaction_name	A string expression that names the transaction. The string must not contain any spaces.
when	Determines when the function begins to measure the transaction time. The possible values are NOW and ONINPUT. When you select NOW (the default), transaction measurement begins as soon as the function is interpreted. When you select ONINPUT, transaction measurement begins when the first input after the **start_transaction** command is generated.

To indicate a transaction to be analyzed, use the **start_transaction** and **end_transaction** functions. These are inserted immediately before and after the transaction, and enable LoadRunner to measure the time it takes for the transaction to be performed.

Transactions can be nested, but each **end_transaction** statement must be associated with a **start_transaction** statement or it will be interpreted as an illegal command. Remember that for the transaction time to be meaningful, you should include a synchronization function before the end of the transaction.

The transaction time is written once the **end_transaction** statement is interpreted. If you do not end the transaction, then no duration time will be recorded. The next time a **start_transaction** statement with the same transaction name is interpreted, the timing restarts at 0.

Note that each transaction in the Vuser script must be declared at the beginning of the Vuser script. You declare transactions using the **declare_transaction** function. For more information, see Chapter 12, “Measuring System Performance Using Transactions.”

Example

In the following example, the deposit "transaction" measures the time it takes for a Vuser to deposit fifty dollars at an ATM. Once the deposit is completed and returns a value to the variable *status,* the transaction is completed.

```
#Declare the transaction name.
declare_transaction ( "deposit");

# Move mouse to deposit button.
move_locator_abs (127, 198, 0);

# Click left mouse button.
click ("Left");

# Move to amount field.
move_locator_abs (141, 350,0);

# Type in $50.
type ("50");

# Move to the OK button.
move_locator_abs (135, 378, 0);

# Define a Deposit transaction.
start_transaction ("deposit" ,ONINPUT);

# Click on the OK button.
click ("Left");

# Wait for "Done" to appear in the ATM window.
wait_text ("Done", 5, ret_text, ret_index, 0, 0, 500, 500, ret_bbox);

# End Deposit transaction.
if (rc == 0)
    end_transaction ("deposit", PASS);
else
    end_transaction ("deposit", FAIL);
```

Return Values

This function returns 0 if the operation is successful or a non-zero value if the operation fails. For more information, see the Return Values table on page 135.

See Also

declare_transaction, end_transaction

user_data_point

records a user-defined data sample.

user_data_point (*sample_name, value* **);**

sample_name	A string indicating the name of the sample type.
value	The value to be recorded.

This function allows you to record your own data for performance analysis. Each time you want to record a piece of data, use this function to record the sample name, and the value. LoadRunner automatically records the time that the sample is recorded. After scenario execution, you can use LoadRunner's User Defined graph to analyze the results. For more information, see Chapter 12, "Measuring System Performance Using Transactions."

Example

In the following example the user data point checks the CPU every second, and records the result.

```
for (i=0;i<100;i++) {
    cpu_val=cpu_check();
    user_data_point ("cpu", cpu_val);
    sleep(1);
}
```

Return value

This function returns 0 if it succeeds, and -1 if it fails to write the sample data.

See Also

declare_transaction, start_transaction, end_transaction

wait_text

waits for a string to appear in a rectangle at a given location.

wait_text (*pattern*, *timeout* **[** , *ret_text, ret_index, x1, y1, x2, y2, ret_bbox* **]** **);**

pattern	The text that VXRunner waits for. This can be a text or NULL string, or a regular expression. If *pattern* is a NULL string, VXRunner waits for *timeout* if there is any text within the specified rectangle. If there is no text within the specified rectangle, VXRunner returns immediately.
timeout	The number of seconds that VXRunner waits for the text to appear.
ret_text	An output variable that stores the actual string that LoadRunner identified as matching the *pattern*.
ret_index	The index of the subexpression that was matched. If *pattern* is a string *ret_index* will equal one when matched. However, if *pattern* is a regular expression it may include a number of *or* operators. In these cases, *ret_index* contains the index of the matched *or* subexpression. For more information, see below.
x1,y1,x2,y2	The coordinates of a rectangle that encloses the text to be read. The pairs of coordinates designate the two diagonally opposite corners of the rectangle.
ret_bbox	An optional array that describes the exact location of the text string within the enclosed rectangle. The array also follows the format x1, y1, x2, y2.

Logical operators may be included in the *pattern* if it is a regular expression. The logical operator or (\ |) is also supported. For example, the function call:

```
wait_text ("!OK\| Error", 10, ret_text, ret_index);
```

sets the *ret_index* parameter if either the “OK” or “Error” strings are found. The exclamation point is specific to LoadRunner and is not part of the

regular expression. If the OK string is found, the *ret_text* is assigned the string "OK", and *ret_index* is assigned the value 1. If the "Error" string is found, *ret_text* is assigned the string "**Error**", and *ret_index* is assigned the value 2.

The **wait_text** function is often used in conjunction with the **expect_text** function. The **expect_text** function instructs VXRunner to ignore all the text currently displayed in the active window and wait for the string defined in the **wait_text** statement to appear. For more information about these two functions, see Chapter 13, "Emulating Server Load: Rendezvous Points."

Example

In the following example, the deposit transaction is defined to measure how long it takes for a Vuser to deposit fifty dollars using the ATM application. The **expect_text** statement instructs VXRunner to ignore all strings currently displayed in the ATM window. The **wait_text** function instructs VXRunner to wait for the "Done" message to appear. When the message appears script replay is resumed and the time taken to perform the deposit transaction is recorded.

```
# Find the window id.
win_find ("ATM", 210, 100, win_no);

# Ignore the text in the ATM window.
expect_text ( );

# Mouse pointer moved to deposit button.
move_locator_abs (10, 10, 0);

# Start measuring deposit operation.
start_transaction ("deposit", ONINPUT);

# Click left mouse button on deposit button.
click ("Left");

# Wait for "Done" to appear in the ATM window.
rc = wait_text ("Done", 5, ret_text, ret_index, 0, 0, 500, 500, ret_bbox);

# End Deposit transaction.
if (rc == 0)
    end_transaction ("deposit", PASS);
```

```
else
    end_transaction ("deposit", FAIL);
```

Return Values

This function returns 0 if the operation is successful or a non-zero value if the operation fails. For more information, see the Return Values table on page 135.

See Also

expect_text, start_transaction, end_transaction

Part VII

Appendices

A

VXRunner Configuration Files

In the VXRunner configuration files, you assign values to the parameters which affect specific VXRunner functions. This appendix describes the VXRunner configuration parameters—including their syntax and default values.

About VXRunner Configuration Files

You can set three levels of configuration files for VXRunner:

- The **bottom-level** configuration file is *vrunner.cfg*. This is a system-wide configuration file which is created by the LoadRunner installation program and which is normally maintained by the system administrator. The file resides in the directory $M_LROOT/dat.

  Included with *vrunner.cfg* is *machine.cfg*—a file which holds all platform-dependent items (such as keyboard configuration). This file is automatically copied to the correct machine version by the LoadRunner installation script.

- The **middle-level** configuration file (optional) is *vrunner.cfg*. A system environment variable, XR_CFG_FILE, designates the location of this file. This file can be used to set values specific to a group of users testing the same application.

- The **top-level** configuration file (optional) is the *.vrunner* file stored in your home directory. This file can be used to tailor VXRunner to your individual needs.

  When you invoke VXRunner, these configuration files are loaded in the order listed above. If the same system variable is assigned a value by more than one of these files, the value set for this variable is the one specified in the highest level configuration file.

Configuration Parameters

This section describes the function of each of the system parameters.

File Storage Parameters

XR_FILE_LOCKING = {TRUE|FALSE}
Activates VXRunner file locking.

(Default = **TRUE**)

XR_SEARCH_PATH = ***directory pathname(s)***
Sets a search path for the tests stored in your LoadRunner database. Any test stored in a directory specified by this parameter can be opened or called using its private name.

Each directory must be designated by its full logical pathname. A space serves as the delimiter between the pathnames of two different directories. The order in which directories are specified determines the order in which they are checked for the specified test.

Note that the search path must be set separately for the Scenario Script window and for the Vuser Development Environment. In the Scenario Script window, the search path only affects calls to other TSL scripts, and does not affect the location of Vuser Test scripts. The full path of Vuser test scripts must be specified.

(Default = Current Directory and **$M_LROOT/lib**)

XR_TMPDIR = ***pathname***
Sets the directory in which LoadRunner will store temporary tests. Each such test is assigned a name having the format: *nonam********, where each asterisk is a letter or digit. These tests are created when you select File > New, and are automatically deleted unless explicitly saved. The directory designated by this parameter should have at least 5 megabytes of storage available for these temporary tests.

(Default = **/tmp**)

Input Device Parameters

XR_INP_KBD_NAME = ***file pathname***
Designates the path and name of the keyboard definition file. This file specifies what string will be generated in the TSL script when each key of the system keyboard is pressed.

(Default = platform-dependent. Specified within the *machine.cfg* file.)

XR_INP_MKEYS = ***mouse_button_code string***
Assigns a unique name (*string*) to each of the mouse buttons. When a test is recorded, this name is the expression enclosed in the TSL **mtype** statement generated whenever the specified button is activated. For example, the default names assigned to each of the three mouse buttons (when pressed alone as well as in conjunction with the SHIFT key) are as follows:

```
XR_INP_MKEYS = x01 Right S_Right \
               x02 Middle S_Middle \
               x04 Left   S_Left
```

Note that button codes are specified here in hexadecimal notation. When defining your mouse keys, be sure to use hexadecimal notation.

Record/Replay Command Softkeys

XR_SOFT_ABORT = ***softkey***
Defines the ABORT softkey. Pressing this softkey is equivalent to selecting Replay > Abort.

(Default is platform-dependent. See the *Installing LoadRunner* guide.)

XR_SOFT_ANIMATE = ***softkey***
Defines the ANIMATE softkey. Pressing this softkey is equivalent to selecting Replay > Animate.

(Default is platform-dependent. See the *Installing LoadRunner* guide.)

XR_SOFT_BREAKPOINT = ***softkey***
Defines the BREAKPOINT softkey. Pressing this softkey is equivalent to selecting Replay > Breakpoint.

(Default is platform-dependent. See the *Installing LoadRunner* guide.)

XR_SOFT_MARKLOCATOR = ***softkey***
Defines the MARK LOCATOR softkey used to record the absolute coordinate position (in pixels) of the screen pointer.

(Default is platform-dependent. See the *Installing LoadRunner* guide.)

XR_SOFT_PAUSE = ***softkey***
Defines the PAUSE softkey. Pressing this softkey is Equivalent to selecting Replay > Pause.

(Default is platform-dependent. See the *Installing LoadRunner* guide.)

XR_SOFT_RECORD = ***softkey***
Defines the RECORD softkey. Pressing this softkey is Equivalent to selecting Replay > Record.

(Default is platform-dependent. See the *Installing LoadRunner* guide.)

XR_SOFT_RUN = ***softkey***
Defines the RUN softkey. Pressing this softkey is equivalent to selecting Replay > Run.

(No default defined)

XR_SOFT_STEP = ***softkey***
Defines the STEP softkey. Pressing this softkey is equivalent to selecting Replay > Step.

(Default is platform-dependent. See the *Installing LoadRunner* guide.)

XR_SOFT_STEP_INTO = ***softkey***
Defines the STEP INTO softkey. Pressing this softkey is equivalent to selecting Replay > Step Into.

(Default is platform-dependent. See the *Installing LoadRunner* guide.)

Synchronization Softkeys

XR_SOFT_LOCATOR_WAIT_REDRAW = ***softkey***
Defines the LOCATOR WAIT REDRAW softkey.

(Default is platform-dependent. See the *Installing LoadRunner* guide.)

XR_SOFT_WAIT_REDRAW = ***softkey***
Defines the WAIT REDRAW softkey.

(Default is platform-dependent. See the *Installing LoadRunner* guide.)

XR_SOFT_WAIT_STRING = ***softkey***
Defines the WAIT STRING softkey.

(Default is platform-dependent. See the *Installing LoadRunner* guide.)

Test Execution Parameters

XR_CLICK_DELAY = ***integer***
Sets the interval, in tenths of a second, that VXRunner waits after inputting a single click during replay. The value assigned to this parameter in the system configuration file can be overridden using Click Delay in the Controls dialog box or the **setvar** TSL function.

(Default = **3** [tenths of a second])

XR_DBLCLK_TIME = ***integer***
Defines the maximum permitted interval, in tenths of a second, that can elapse between two clicks that constitute a double-click. The value assigned to this parameter in the system configuration file can be overridden using Double Click Time in the Controls dialog box or the **setvar** TSL function. The minimum value is 10 (tenths of a second).

(Default = **10** [tenths of a second])

XR_FAST_REPLAY = {TRUE|FALSE}
Sets the default value for the Fast Replay check box in the Controls dialog box. The value assigned to this parameter in the system configuration file can be overridden using the Fast Replay check box in the Controls dialog box or using the **setvar** TSL function. When you select this option, the Synchronize option is automatically turned on.

(Default = **FALSE**, regular replay)

XR_FOCUS_DELAY = ***integer***
Defines the interval, in tenths of a second, that VXRunner waits for a window to come into focus when that window becomes the active window. The value assigned to this parameter in the system configuration file can be

overridden using the Focus Delay check box in the Controls dialog box or the **setvar** TSL function.

(Default = **3** [tenths of a second])

XR_KEY_EDITING = {TRUE|FALSE}
Activates or deactivates key editing. When activated, VXRunner generates TSL type statements that are more concise. The value assigned to this parameter in the system configuration file can be overridden using the Key Editing check box in the Controls dialog box.

(Default = **TRUE**)

XR_KBD_DELAY = *integer*
Sets the interval, in tenths of a second, that XRunner waits after inputting a single keyboard event during replay.

(Default = 0)

XR_MOVE_WINDOWS = {TRUE|FALSE}
Determines whether, after opening a window at some different location during test replay, VXRunner will automatically move the window to the position recorded in the TSL script. The value assigned to this parameter in the system configuration file can be overridden using the Move Windows check box in the Controls dialog box.

Note: If you set this parameter to FALSE, you will have to take measures to ensure that, during replay, windows are opened in the correct, previously-recorded position.

(Default = **TRUE**)

XR_RAISE_WINDOWS = {TRUE|FALSE}
Sets whether LoadRunner will automatically raise a moved window to the front of the screen display. The value assigned to this parameter in the system configuration file can be overridden using the Raise Windows check box in the Controls dialog box.

(Default = **TRUE**)

XR_RETRY_DELAY = ***integer***
Sets the interval the VXRunner will wait for a window to be silent before considering it fully redrawn and entering input. The value assigned to this parameter in the system configuration file can be overridden using Delay in the Controls dialog box or the **setvar** TSL function.

(Default = **2** [seconds])

XR_SPEED_RANGE = ***integer***
Sets the outer limits (minimum and maximum speeds) displayed on the speed bar control. When the default value (5) is active, you can adjust the replay speed from one fifth to five times the speed at which the test was recorded. (Activating Fast Replay will play back the test at the fastest rate possible.) Note: If the value you enter for this parameter is too high, events may be lost during test execution.

(Default = **5**)

XR_SYNC_TIME = ***integer***
Determines the maximum amount of time (in seconds) that the system waits for an expected synchronization event before giving up and continuing execution of the test. If this time is exceeded, replay continues after a slight delay. The value assigned to this parameter in the system configuration file can be overridden using Sync Time in the Controls dialog box or the **setvar** TSL function.

(Default = **10** [seconds])

XR_SYNCHRONIZED = **{TRUE|FALSE}**
Sets whether test execution will utilize synchronization data stored in the test database. If you set this parameter to FALSE, the reliability of test execution may be impaired. The value assigned to this parameter in the system configuration file can be overridden using the Synchronize check box in the Controls dialog box or the **setvar** TSL function.

(Default = **TRUE**)

XR_TIMEOUT = ***integer***
Sets the global timeout (in seconds) used by LoadRunner. This value is added to the *time* parameter imbedded in **wait_window** statements to determine the maximum amount of time that VXRunner will search for the specified window.

(Default = **30** [seconds])

Script Display Parameters

XR_INSERT_NEWLINES = {TRUE|FALSE}
Sets whether or not VXRunner will insert a blank line before and after **check_window** and **wait_window** statements.

(Default = **TRUE**)

XR_EDITOR_MAX_CHARS = ***integer***
Determines the maximum number of characters that can be written per line in the VXRunner window. When, during a Record session, a generated script statement extends beyond this maximum length, the script line is split between two or more lines.

(Default = **80**)

Text Editor Parameter

XR_TEXTEDIT = ***text editor name***
Specifies the text editor in which execution reports will be loaded. By default, this parameter is commented. It should be uncommented only if a text editor other than the OpenWindows *textedit* program is to be used.

In such a case set this parameter so that it points to the name of a script which will activate your editor. This script should receive two parameters: (1) the name of the X display to use and (2) the name of the file to be edited. The script should invoke the user-specified editor. Note that the referenced script must open a window and execute the command in the background.

(Default = **textedit**)

Configuration File Contents

System configuration files are text files that can include the following types of data:

- assignment statements
- directives

- blank lines
- comments

Assignment Statements

The main purpose of a configuration file is to allow the assignment of values to VXRunner system parameters. The values assigned to these parameters determine how the program will run. Most of these values will be defined by the top-level (*~/.vrunner*) configuration file located in the *$M_ROOT/dat* directory following system installation. Other parameters point to system locations.

In addition to assignment statements used to set values for parameters, you can assign values to user-defined variables. A typical use would be to assign an arbitrary shorthand name to a path which may appear any number of times in the configuration file. For example, the line

```
P153T = /project_15/ver_3/tests
```

assigns the specified pathname to the variable P153T. Whenever the name of this variable subsequently appears in the configuration file, the associated path will be understood. Thus if the location of the search path is specified by the line

```
XR_search_path = $(P153T)/..
```

LoadRunner will understand that this directory resides under the pathname **/project_15/ver_3**.

Note the following points:

- The equal sign (=) is always used to assign a value to a variable, whether this variable is a system parameter or a user-defined variable.
- Whenever a user-defined variable is initiated in the file, the name of this variable must be enclosed within parentheses; the enclosed variable name is preceded by a dollar sign ($). Note that environmental variables may also be accessed using the same convention.
- If the same item appears more than once in the configuration file, the *last* value assigned to this item will be used by the system.

Regarding case sensitivity of names, note the following points:

- Names of variables and system parameters appearing in the configuration file are *not* case sensitive.
- Boolean values assigned to variables or parameters are *not* case sensitive.
- Values assigned to certain system variables *may* be case sensitive, depending on the nature of the variable.

Directives

The configuration file can contain one or more **include** directives. An include directive is used to integrate the entire contents of the specified file in the configuration data processed by LoadRunner. An **include** directive consists of:

- the at sign @ in the first column of the file line
- the label **include**
- the @ character, followed by the name of the file (enclosed between quotation marks) to be integrated at this point. For example:

```
@include "@loc_file"
.
.
@include "@/file2"
```

Note that each file to be integrated in the configuration data must be specified by its own **include** entry.

Include directives can be nested: A file that is referenced by an **include** entry in the configuration file may in turn contain its own **include** directives to other files. Such nesting is supported up to ten levels. When a *relative* name is used to specify the file to be integrated, the specified name must express the location of this file relative to the file in which the calling **include** directive appears.

Blank Lines and Comments

As it processes a configuration file, LoadRunner ignores blank lines and comments. Comments may be inserted in a file using the number sign (#). All text that appears between a number sign and the end of a line is understood to be a comment.

Line Format

When a record in the configuration file extends beyond a single line, the backslash character (\) indicates that the record continues on the next line.

```
XR_INP_MKEYS = 0 x01 Right S_Right \
    0 x02 Middle S_Middle \
    0 x04 Left   S_Left
```

In the above example, the XR_INP_MKEYS parameter is used to assign a unique name (string) to each of the mouse buttons when pressed alone as well as in conjunction with the SHIFT key.

When more than one value is assigned to the same parameter or user-defined variable, the delimiter between the values is a blank space.

Special Characters

The backslash character (\) can also be used within a configuration file as an escape character. If a record must include a special character which has a different, reserved function, precede the special character with a backslash. The character that follows will then be read literally by the LoadRunner interpreter. (For example, in order that a backslash be understood as a literal backslash, type in a double backslash [\\].)

The backslash character is also used to indicate literal carriage return (ENTER) and tab characters in the configuration file:

Quotation marks (") can be used to indicate that two or more string segments constitute a single value.

B

Command Softkeys

When you are recording and replaying GUI Vuser scripts, it is often convenient to use softkeys instead of selecting menu options with the mouse. The table below shows the default softkey combinations for the Sun, IBM, HP, and DEC platforms.

Command	SUN	HP	IBM	DEC
RECORD	**F4**	**F4**	**F4**	**F4**
ANIMATE	**F8**	**F8**	**F8**	**F8**
RUN	Unbound	Unbound	Unbound	Unbound
STEP	**F7**	**F7**	**F7**	**F7**
STEP INTO	**F9**	**Ctrl_L+F7**	**F9**	**Ctrl_L+F7**
ABORT/ STOP	**STOP**	**F10**	**F1**	**Alt_L+F1**
PAUSE	**PAUSE**	**F11**	**F10**	**F10**
BREAKPOINT	**F5**	**F5**	**F5**	**Alt_L+F7**
MARK LOCATOR	**F6**	**F6**	**F6**	**F6**
WAIT WINDOW	**F3**	**F3**	**F3**	**F2**
GET TEXT	**Shift_L + F6**	**Shift_L + F6**	**Shift_L + F6**	**F11**
WAIT TEXT	**Shift_L + F5**	**Shift_L + F5**	**Shift_L + F5**	**F1**

Softkey combinations are configurable. If any of the default combinations interfere with a softkey that another application uses, you can reconfigure the LoadRunner command to another combination. For details, refer to the *Installing LoadRunner* guide.

Index

D

E

F

G

H

I

K

L

M

O

P

R

S

T

U

V

W

X

Mercury Interactive Corporation
1325 Borregas Avenue
Sunnyvale, CA 94089 USA
Main Telephone: (408) 822-5200
Sales & Information: (800) TEST-911
Customer Support: (877) TEST-HLP
Fax: (408) 822-5300
Home Page: www.mercuryinteractive.com
Customer Support: support.mercuryinteractive.com