MERCURY WINRUNNER™

VERSION 8.2

Advanced Features User's Guide

MERCURY™

Mercury WinRunner

Advanced Features User's Guide

Version 8.2

MERCURY™

Mercury WinRunner Advanced Features User's Guide, Version 8.2

Mercury Interactive Corporation
379 North Whisman Road
Mountain View, CA 94043
Tel: (650) 603-5200
Toll Free: (800) TEST-911
Customer Support: (877) TEST-HLP
Fax: (650) 603-5300

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

WRAG8.2/01

Multi-Volume Chapter Summary

WinRunner user documentation is divided into two volumes:

- The *Mercury WinRunner Basic Features User's Guide* introduces WinRunner and describes its mainstream features and automated testing procedures.
- The *Mercury WinRunner Advanced Features User's Guide* describes WinRunner's advanced features, introduces Mercury's Test Script Language (TSL), and covers advanced configuration options. It also describes how to integrate WinRunner with other Mercury products.

A listing of the chapters in each guide is provided below:

Mercury WinRunner Basic Features User's Guide

PART I: STARTING THE TESTING PROCESS

PART II: INTRODUCING THE GUI MAP

PART III: CREATING TESTS—BASIC

PART IV: RUNNING TESTS—BASIC

PART V: CONFIGURING BASIC SETTINGS

Mercury WinRunner Advanced Features User's Guide

PART I: WORKING WITH THE GUI MAP

PART II: CREATING TESTS—ADVANCED

PART III: PROGRAMMING WITH TSL

PART IV: RUNNING TESTS—ADVANCED

PART V: DEBUGGING TESTS

PART VI: CONFIGURING ADVANCED SETTINGS

PART VII: WORKING WITH OTHER MERCURY PRODUCTS

Table of Contents

PART I: WORKING WITH THE GUI MAP

PART II: CREATING TESTS—ADVANCED

PART III: PROGRAMMING WITH TSL

PART IV: RUNNING TESTS—ADVANCED

PART V: DEBUGGING TESTS

PART VI: CONFIGURING ADVANCED SETTINGS

PART VII: WORKING WITH OTHER MERCURY PRODUCTS

Welcome to Mercury WinRunner

Welcome to WinRunner, the Mercury enterprise test automation solution. With WinRunner you can create and run sophisticated automated tests on your application.

Note: The *Mercury WinRunner Basic Features User's Guide* and *Mercury WinRunner Advanced Features User's Guide* are available as separate books only in the printed version. In the PDF and context-sensitive Help, the information is combined.

Using this Guide

This guide describes the main concepts behind automated software testing. It provides step-by-step instructions to help you create, debug, and run tests, and to report defects detected during the testing process.

The *Mercury WinRunner Basic Features User's Guide* provides detailed descriptions of WinRunner's features and automated testing procedures. The *Mercury WinRunner Advanced Features User's Guide* describes WinRunner's advanced features. It is recommended that users of the *Mercury WinRunner Advanced Features User's Guide* have a working knowledge of the information covered in the *Mercury WinRunner Basic Features User's Guide*.

This guide contains the following parts:

Part I **Working with the GUI Map**

Describes how to merge and configure GUI map files. It also describes how to teach WinRunner to recognize bitmaps as GUI objects by defining bitmaps as *virtual objects*.

Part II **Creating Tests—Advanced**

Describes how to use regular expressions, and handle unexpected events that occur during a test run.

Part III **Programming with TSL**

Describes how to enhance your test scripts using variables, control-flow statements, arrays, user-defined and external functions, Runner's visual programming tools, and interactive input during a test run.

Part IV **Running Tests—Advanced**

Describes how to run batch tests, and how to run tests both from within WinRunner and from the command line.

Part V **Debugging Tests**

Describes how to control test runs to identify and isolate bugs in test scripts, by using breakpoints and monitoring variables during the test run.

Part VI **Configuring Advanced Settings**

Describes how to customize WinRunner's user interface, test script editor and the Function Generator. It also describes how to initialize special configurations to adapt WinRunner to your testing environment.

Part VII **Working with Other Mercury Products**

Describes how to integrate WinRunner with QuickTest Professional, Quality Center, Business Process Testing, and LoadRunner.

WinRunner Documentation Set

In addition to this Advanced Features User's Guide, WinRunner comes with a complete set of printed documentation:

WinRunner Basic Features User's Guide provides step-by-step instructions for using WinRunner to meet the special testing requirements of your application.

WinRunner Installation Guide describes how to install WinRunner on a single computer or a network.

WinRunner Tutorial teaches you basic WinRunner skills and shows you how to start testing your application.

TSL Reference Guide describes the WinRunner Test Script Language (TSL) and the functions it contains.

WinRunner Customization Guide explains how to customize WinRunner to meet the special testing requirements of your application.

Online Resources

WinRunner includes the following online resources, accessible from the program group or Help menu:

Read Me provides last-minute news and information about WinRunner.

WinRunner Help provides immediate context-sensitive answers to questions that arise as you work with WinRunner. It describes menu commands and dialog boxes, and shows you how to perform WinRunner tasks.

WinRunner Quick Preview provides a short presentation of the main WinRunner capabilities for new WinRunner users.

TSL Online Reference describes the WinRunner Test Script Language (TSL), the functions it contains, and examples of how to use the functions.

Printer-Friendly Documentation displays the complete documentation set in PDF format. The printer-friendly books can be read and printed using Adobe Acrobat Reader. It is recommended that you use version 5.0 or later. You can download the latest version of Adobe Acrobat Reader from www.adobe.com.

Sample Tests includes utilities and sample tests with accompanying explanations.

What's New in WinRunner describes the newest features in the latest versions of WinRunner.

Note: The Mercury WinRunner User's Guide online version is a single volume, while the printed and PDF versions consists of two books, the *Mercury WinRunner Basic Features User's Guide* and the *Mercury WinRunner Advanced Features User's Guide*.

Technical Support Online uses your default Web browser to open the Mercury Customer Support Web site. The URL for this Web site is http://support.mercury.com.

Mercury Interactive on the Web uses your default web browser to open Mercury Interactive's home page. This site provides you with the most up-to-date information on Mercury Interactive, its products and services. This includes new software releases, seminars and trade shows, customer support, training, and more. The URL for this Web site is http://www.mercury.com.

Documentation Updates

Mercury is continuously updating its product documentation with new information. You can download the latest version of this document from the Customer Support Web site (http://support.mercury.com).

To download updated documentation:

1. In the Customer Support Web site, click the **Documentation** link.
2. Under **Select Product Name**, select **WinRunner**.

   Note that if **WinRunner** does not appear in the list, you must add it to your customer profile. Click **My Account** to update your profile.
3. Click **Retrieve**. The Documentation page opens and lists the documentation available for the current release and for previous releases. If a document was recently updated, **Updated** appears next to the document name.
4. Click a document link to download the documentation.

Typographical Conventions

This book uses the following typographical conventions:

1, 2, 3 Bold numbers indicate steps in a procedure.

> The greater-than sign separates menu levels (for example, **File** > **Open**).

Stone Sans The **Stone Sans** font indicates names of interface elements (for example, the **Run** button) and other items that require emphasis.

Bold **Bold** text indicates method or function names.

Italics *Italic* text indicates method or function arguments, file names in syntax descriptions, and book titles. It is also used when introducing a new term.

<> Angle brackets enclose a part of a file path or URL address that may vary from user to user (for example, <**MyProduct installation folder**>\bin).

Arial The Arial font is used for examples and text that is to be typed literally.

Arial bold The **Arial bold** font is used in syntax descriptions for text that should be typed literally.

Small Caps The Small Caps font indicates keyboard keys.

... In a line of syntax, an ellipsis indicates that more items of the same format may be included. In a programming example, an ellipsis is used to indicate lines of a program that were intentionally omitted.

[ ] Square brackets enclose optional arguments.

| A vertical bar indicates that one of the options separated by the bar should be selected.

Part I

Working with the GUI Map

1

Merging GUI Map Files

This chapter explains how to merge GUI map files. This is especially useful if you have been working in the *GUI Map File per Test* mode and want to start working in the *Global GUI Map File* mode. It is also useful if you want to combine GUI map files while working in the *Global GUI Map File* mode.

This chapter describes:

- About Merging GUI Map Files
- Preparing to Merge GUI Map Files
- Resolving Conflicts while Automatically Merging GUI Map Files
- Merging GUI Map Files Manually
- Changing to the GUI Map File per Test Mode

About Merging GUI Map Files

When you work in the *GUI Map File per Test* mode, WinRunner automatically creates, saves, and loads a GUI map file with each test you create. This is the simplest way for beginners to work in WinRunner. It is not the most efficient, however. When you become more familiar with WinRunner, you may want to change to working in the *Global GUI Map File* mode. This mode is more efficient, as it enables you to save information about the GUI of your application in a GUI map that is referenced by several tests. When your application changes, instead of updating each test individually, you can merely update the GUI map that is referenced by an entire group of tests.

The GUI Map File Merge Tool enables you to merge multiple GUI map files into a single GUI map file. Before you can merge GUI map files, you must specify at least two source GUI map files to merge and at least one GUI map file as a target file. The target GUI map file can be an existing file or a new (empty) file.

You can work with this tool in either automatic or manual mode.

- When you work in automatic mode, the merge tool merges the files automatically. If there are conflicts between the merged files, the conflicts are highlighted and you are prompted to resolve them.
- When you work in manual mode, you must add GUI objects to the target file manually. The merge tool does not highlight conflicts between the files.

In both modes, the merge tool prevents you from creating conflicts while merging the files.

Once you merge GUI map files, you must also change the GUI map file mode, and modify your tests or your startup test to load the appropriate GUI map files.

Preparing to Merge GUI Map Files

Before you can merge GUI map files, you must decide in which mode to merge your files and specify the source files and the target file.

To start merging GUI map files:

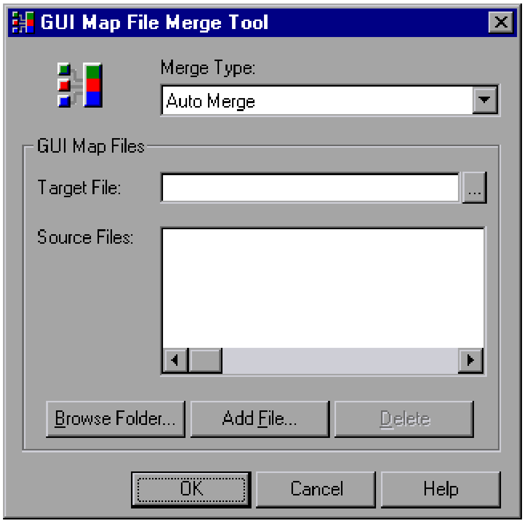
1 Choose **Tools** > **Merge GUI Map Files**.

A WinRunner message box informs you that all open GUI maps will be closed and all unsaved changes will be discarded.

2 To continue, click **OK**.

To save changes to open GUI maps, click **Cancel** and save the GUI maps using the GUI Map Editor. For information on saving GUI map files, refer to Chapter 7, “Editing the GUI Map” in the *Mercury WinRunner Basic Features User’s Guide*. Once you have saved changes to the open GUI map files, start again at step 1.

The GUI Map File Merge Tool opens, enabling you to select the merge type and specify the target files and source file.



Merge Type: Auto Merge

3 In the **Merge Type** box, accept **Auto Merge** or select **Manual Merge**.

- **Auto Merge** merges the files automatically. If there are conflicts between the merged files, the conflicts are highlighted and you are prompted to resolve them.
- **Manual Merge** enables you to manually add GUI objects from the source files to the target file. The merge tool does not highlight conflicts between the files.

4 To specify the target GUI map file, click the browse button opposite the **Target File** box. The Save GUI File dialog box opens.

- To select an existing GUI map file, browse to the file and highlight it so that it is displayed in the File name box. When prompted, click **OK** to replace the file.
- To create a new (empty) GUI map file, browse to the desired folder and enter the name of a new file in the **File name** box.

5 Specify the source GUI map files.

- To add all the GUI map files in a folder to the list of source files, click the **Browse Folder** button. The Set Folder dialog box opens. Browse to the desired folder and click **OK**. All the GUI map files in the folder are added to the list of source files.
- To add a single GUI map file to the list of source files, click the **Add File** button. The Open GUI File dialog box opens. Browse to the desired file and highlight it so that it is displayed in the File name box and click **OK**.
- To delete a source file from the list, highlight a GUI map file in the **Source Files** box and click **Delete**.

6 Click **OK** to close the dialog box.

- If you chose **Auto Merge** and the source GUI map files are merged successfully without conflicts, a message confirms the merge.
- If you chose **Auto Merge** and there are conflicts among the source GUI map files being merged, a WinRunner message box warns of the problem. When you click OK to close the message box, the GUI Map File Auto Merge Tool opens. For additional information, see "Resolving Conflicts while Automatically Merging GUI Map Files" on page 6.
- If you chose **Manual Merge**, the GUI Map File Manual Merge Tool opens. For additional information, see "Merging GUI Map Files Manually" on page 10.

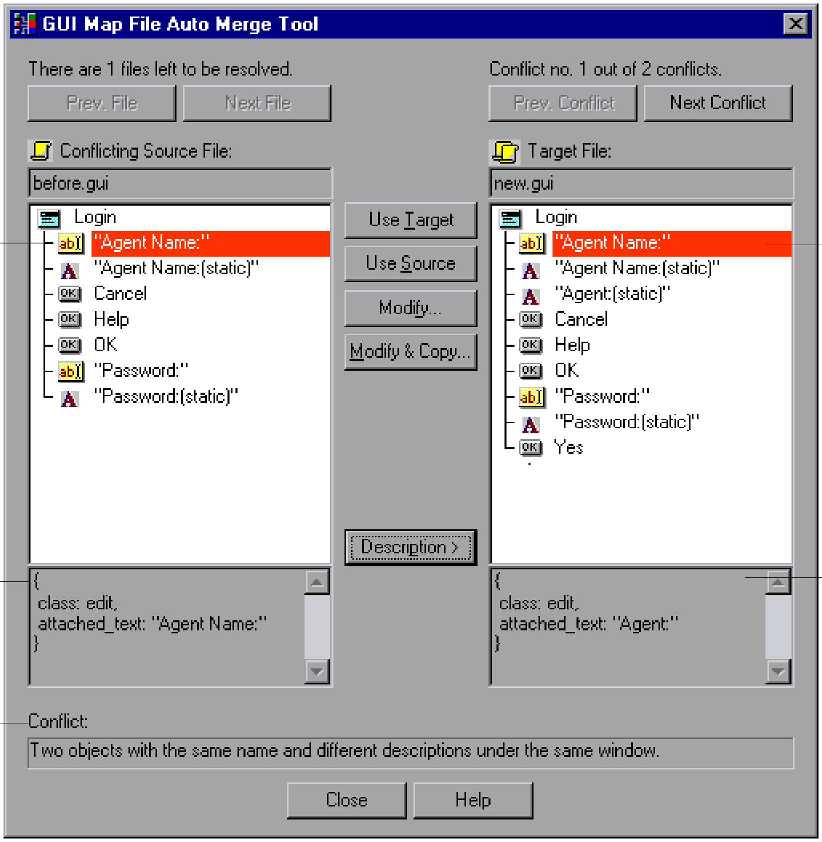
Resolving Conflicts while Automatically Merging GUI Map Files

If you chose the **Auto Merge** option in the GUI Map File Merge Tool and there were no conflicts between files, then a message confirms the merge.

When you merge GUI map files automatically, conflicts occur under the following circumstances:

- Two windows have the same name but different physical descriptions.
- Two objects in the same window have the same name but different physical descriptions.

The following example demonstrates automatically merging two conflicting source files (*before.gui* and *after.gui*) into a new target file (*new.gui*). The GUI Map File Auto Merge Tool opens after clicking OK in the conflict warning message box, as described in “Preparing to Merge GUI Map Files” on page 4. It enables you to resolve conflicts and prevents you from creating new conflicts in the target file.



There are 1 files left to be resolved.

Conflict no. 1 out of 2 conflicts.

Conflicting Source File:

before.gui

```
Login
  "Agent Name:"
  "Agent Name:(static)"
  Cancel
  Help
  OK
  "Password:"
  "Password:(static)"
```

```
{
class: edit,
attached_text: "Agent Name:"
}
```

Target File:

new.gui

```
Login
  "Agent Name:"
  "Agent Name:(static)"
  "Agent:(static)"
  Cancel
  Help
  OK
  "Password:"
  "Password:(static)"
  Yes
```

```
{
class: edit,
attached_text: "Agent:"
}
```

Conflict:

Two objects with the same name and different descriptions under the same window.

Logical name of conflicting object

Logical name of conflicting object

Physical description of conflicting object

Physical description of conflicting object

Description of conflict

The conflicting objects are highlighted in red and the description of the conflict appears in a pane at the bottom of the dialog box. The files conflict because in both GUI map files, there is an object under the same window with the same name and different descriptions. Note that the windows and objects from the *after.gui* source file were copied to the *new.gui* target file without conflicts, since the *new.gui* file was initially empty. The names of the conflicting objects are displayed in red.

The source files are merged in the order in which they appear in the GUI Map File Merge Tool, as described in "Preparing to Merge GUI Map Files" on page 4.

To view the physical description of the conflicting objects or windows, click **Description**.

Each highlighted conflict can be resolved by clicking any of the following buttons. Note that these buttons are enabled only when the conflicting object/window is highlighted in both panes:

Conflict Resolution Option	Description
Use Target	Resolves the conflict by using the name and physical description of the object/window in the target GUI map file.
Use Source	Resolves the conflict by using the name and physical description of the object/window in the source GUI map file.
Modify	Resolves the conflict by suggesting a regular expression (if possible) for the physical description of the object/window in the target GUI map file that will describe both the target and the source object/window accurately. You can modify this description.
Modify & Copy	Resolves the conflict by enabling you to edit the physical description of the object/window in the source GUI map file in order to paste it into the target GUI map file. **Note:** Your changes to the physical description are not saved in the source GUI map file.

Tips:

If there are multiple conflicting source files, you can click **Prev. File** or **Next File** to switch between current GUI map files.

If there are multiple conflicting objects within a single source file, you can click **Prev. Conflict** or **Next Conflict** to switch between highlighted conflicts. If you use your mouse to highlight a non-conflicting object in the target file (for example, to see its physical description) and no conflict is highlighted in the target file, you can click **Prev. Conflict** to highlight the conflicting object.

Once all conflicts between the current source file and the target file have been resolved, the source file is automatically closed and the next conflicting source file is opened. Once all conflicts between GUI map files have been resolved, the remaining source file and the target file are closed, and the GUI Map File Auto Merge Tool closes.

Tip: Sometimes, all the conflicts in the current source file will have been resolved as a result of resolving conflicts in other source files, for example, when modifying an object in the target file that used to conflict with objects other than the current one. When this happens, the **Remove File** button is displayed. Click this button to remove the current source file from the list of source GUI map files.

Note: Changes to the target GUI map file are saved automatically.

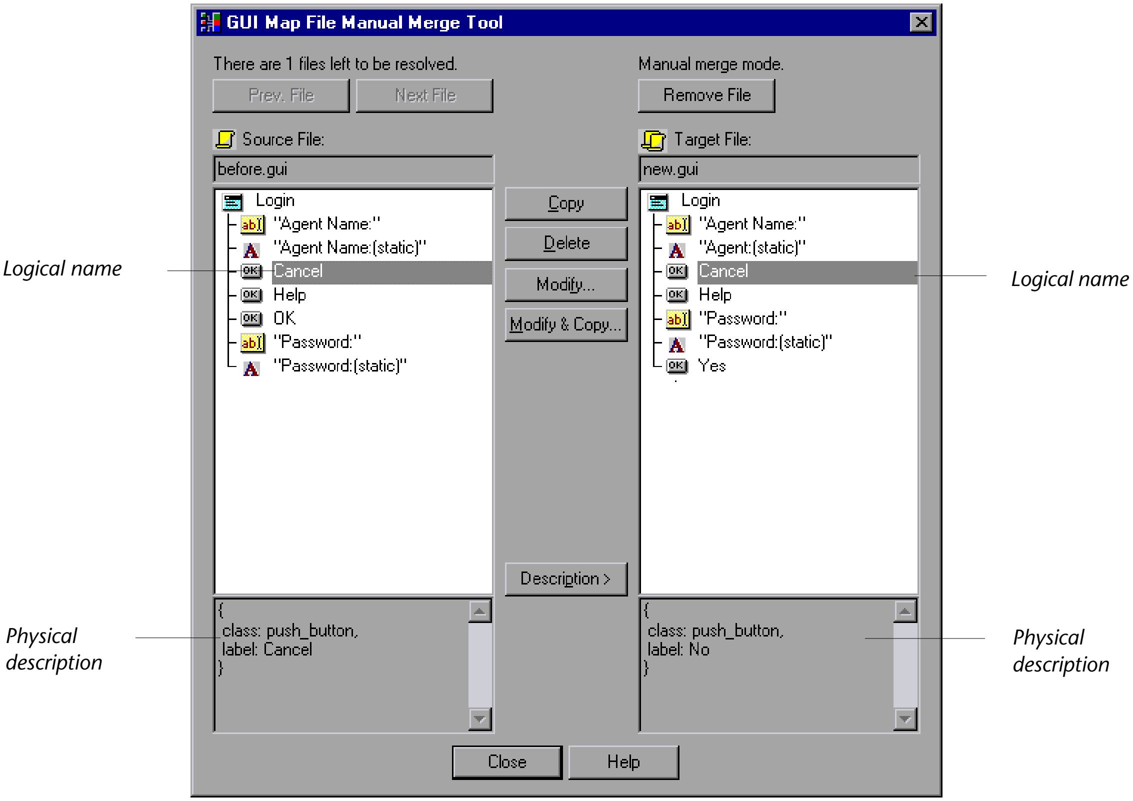
Merging GUI Map Files Manually

When you merge GUI map files manually, you merge each target file with the source file. The merge tool prevents you from creating conflicts while merging the files.

When you merge GUI map files manually, the target GUI map file cannot contain any of the following:

- Two windows with the same name but different physical descriptions.
- Two windows with the same name and the same physical descriptions (identical windows).
- Two objects in the same window with the same name but different physical descriptions.
- Two objects in the same window with the same name and the same physical descriptions (identical objects).

In the following example, the entire contents of the *after.gui* source file was copied to the *new.gui* target file, and there are conflicts between the *before.gui* source file and the target file:



GUI Map File Manual Merge Tool

There are 1 files left to be resolved.

Manual merge mode.

Source File: before.gui

```
Login
├ "Agent Name:"
├ "Agent Name:(static)"
├ Cancel
├ Help
├ OK
├ "Password:"
└ "Password:(static)"
```

```
{
class: push_button,
label: Cancel
}
```

Target File: new.gui

```
Login
├ "Agent Name:"
├ "Agent:(static)"
├ Cancel
├ Help
├ "Password:"
├ "Password:(static)"
└ Yes
```

```
{
class: push_button,
label: No
}
```

Note that in the above example, the highlighted objects in both panes have identical logical names but different descriptions. Therefore, they cannot both exist "as is" in the merged file.

To merge GUI map files manually:

1 Follow the procedure described in "Preparing to Merge GUI Map Files" on page 4 and choose **Manual Merge** as the merge type. After you specify the source and target files and click **OK**, the GUI Map File Manual Merge Tool opens.

The contents of the source file and target file are displayed.

2 Locate the windows or objects to merge.

- ➤ You can double-click windows to see the objects in the window.
- ➤ If there are multiple source files, you can click **Prev. File** or **Next File** to switch between current GUI map files.
- ➤ To view the physical description of the highlighted objects or windows, click **Description**.

3 Merge the files using the following merge options:

Merge Option	Description
Copy (enabled only when an object/window in the current source file is highlighted)	Copies the highlighted object/window in source file to the highlighted window or to the parent window of the highlighted object in the target file. **Note:** Copying a window also copies all objects within that window.
Delete (enabled only when an object/window in the target file is highlighted)	Deletes the highlighted object/window from the target GUI map file. **Note:** Deleting a window also deletes all objects within that window.
Modify (enabled only when an object/window in the target file is highlighted)	Opens the Modify dialog box, where you can modify the logical name and/or physical description of the highlighted object/window in the target file.
Modify & Copy (enabled only when an object/window in the current source file is highlighted)	Opens the Modify dialog box, where you can modify the logical name and/or physical description of the highlighted object/window from the source file and copy it to the highlighted window or to the parent window of the highlighted object in the target file. **Note:** Your changes to the physical description are not saved in the source GUI map file.

Note: Your changes to the target GUI map file are saved automatically.

Tips: If you have finished merging a source file, you can click **Remove File** to remove it from the list of source files to merge.

If there are multiple source files, you can click **Prev. File** or **Next File** to switch between current GUI map files.

Changing to the GUI Map File per Test Mode

When you want to change from working in the *GUI Map File per Test* mode to the *Global GUI Map File* mode, the most complicated preparatory work is merging the GUI map files, as described earlier in this chapter.

In addition, you must also make the following changes:

- You should modify your tests or your startup test to load the GUI map files. For information on loading GUI map files, refer to Chapter 5, "Working in the Global GUI Map File Mode" in the *Mercury WinRunner Basic Features User's Guide*.
- You must select **Global GUI Map File** in the **GUI Files** section in the **General** category of the General Options dialog box.

  When you close WinRunner, you will be prompted to save changes made to the configuration. Click **Yes**.

  Note: In order for this change to take effect, you must restart WinRunner.

  For additional information on the General Options dialog box, refer to Chapter 23, "Setting Global Testing Options" in the *Mercury WinRunner Basic Features User's Guide*.
- You should remember to save changes you make to GUI map files once you switch GUI map file modes. For additional information, refer to Chapter 5, "Working in the Global GUI Map File Mode" in the *Mercury WinRunner Basic Features User's Guide*.

2

Configuring the GUI Map

This chapter explains how to change the way WinRunner identifies GUI objects during Context Sensitive testing.

This chapter describes:

- About Configuring the GUI Map
- Understanding the Default GUI Map Configuration
- Mapping a Custom Object to a Standard Class
- Configuring a Standard or Custom Class
- Creating a Permanent GUI Map Configuration
- Deleting a Custom Class
- Understanding WinRunner Objects Classes
- Understanding Object Properties
- Understanding Default Learned Properties
- Properties for Visual Basic Objects
- Properties for PowerBuilder Objects

About Configuring the GUI Map

Each GUI object in the application being tested is defined by multiple properties, such as class, label, MSW_class, MSW_id, x (coordinate), y (coordinate), width, and height. WinRunner uses these properties to identify GUI objects in your application during Context Sensitive testing.

When WinRunner learns the description of a GUI object, it does not learn all its properties. Instead, it learns the minimum number of properties to provide a unique identification of the object. For each object class (such as push_button, list, window, or menu), WinRunner learns a default set of properties: its GUI map configuration.

For example, a standard push button is defined by 26 properties, such as MSW_class, label, text, nchildren, x, y, height, class, focused, enabled. In most cases, however, WinRunner needs only the *class* and *label* properties to create a unique identification for the push button. Occasionally, the property set defined for an object class may not be sufficient to create a unique description for a particular object. In these cases, WinRunner learns the defined property set plus a selector property, which assigns each object an ordinal value based on the object's location compared to other objects with identical descriptions.

If the default set of properties learned for an object class are not ideal for your application, you can configure the GUI map to learn a different set of properties for that class. For example, one of the default properties for an edit box is the *attached_text* property. If your application contains edit boxes without attached text properties, then when recording, WinRunner may capture the attached text property of another object near the edit box and save that value as part of the object description. In this case, you may want to remove the *attached_text* property from the default set of learned properties and add another property instead.

You can also modify the type of selector used for a class or the recording method used.

Many applications also contain custom GUI objects. A custom object is any object not belonging to one of the standard classes used by WinRunner. These objects are therefore assigned to the generic "object" class. When WinRunner records an operation on a custom object, it generates **obj_mouse_** statements in the test script.

If a custom object is similar to a standard object, you can map it to one of the standard classes. You can also configure the properties WinRunner uses to identify a custom object during Context Sensitive testing. The mapping and the configuration you set are valid only for the current WinRunner session.

To make the mapping and the configuration permanent, you must add configuration statements to your startup test script. Each time you start WinRunner, the startup test activates this configuration.

Note: If your application contains owner-drawn custom buttons, you can map them all to one of the standard button classes instead of mapping each button separately. You do this by either choosing a standard button class in the **Record owner-drawn buttons as** box in the Record category in the General Options dialog box or setting the *rec_owner_drawn* testing option with the **setvar** function from within a test script. For more information on the General Options dialog box, refer to Chapter 23, “Setting Global Testing Options” in the *Mercury WinRunner Basic Features User’s Guide*. For more information on setting testing options with the **setvar** function, see Chapter 21, “Setting Testing Options from a Test Script.”

Object properties vary in their degree of portability. Some are non-portable (unique to a specific platform), such as MSW_class or MSW_id. Some are semi-portable (supported by multiple platforms, but with a value likely to change), such as handle, or Toolkit_class. Others are fully portable (such as label, attached_text, enabled, focused or parent).

Note about configuring non-standard Windows objects: You can use the GUI Map Configuration tool to modify how WinRunner recognizes objects with a window handle (HWND), such as standard Windows objects, ActiveX and Visual Basic controls, PowerBuilder objects, and some Web objects. For additional information on which Web objects are supported for the GUI Map Configuration tool, refer to Chapter 10, "Working with Web Objects" in the *Mercury WinRunner Basic Features User's Guide*. If you are working with a WinRunner add-in to test other objects, you can use the GUI map configuration functions, such as **set_record_attr**, and **set_record_method**. For additional information about these functions, refer to the *TSL Reference*. Some add-ins also have their owns tools for configuring how WinRunner recognizes objects in a specific toolkit. For additional information, refer to the *Read Me* file for your WinRunner add-in.

Understanding the Default GUI Map Configuration

For each class, WinRunner learns a set of default properties. Each default property is classified "obligatory" or "optional". (For a list of the default properties, see "Understanding Object Properties" on page 31.)

- An *obligatory* property is always learned (if it exists).
- An *optional* property is used only if the obligatory properties do not provide unique identification of an object. These optional properties are stored in a list. WinRunner selects the minimum number of properties from this list that are necessary to identify the object. It begins with the first property in the list, and continues, if necessary, to add properties to the description until it obtains unique identification for the object.

If you use the GUI Spy to view the default properties of an OK button, you can see that WinRunner learns the class and label properties. The physical description of this button is therefore:

```
{class:push_button, label:"OK"}
```

In cases where the obligatory and optional properties do not uniquely identify an object, WinRunner uses a *selector*. For example, if there are two OK buttons with the same MSW_id in a single window, WinRunner would use a selector to differentiate between them. Two types of selectors are available:

- A *location* selector uses the spatial position of objects.
- An *index* selector uses a unique number to identify the object in a window.

The *location* selector uses the spatial order of objects within the window, from the top left to the bottom right corners, to differentiate among objects with the same description.

The *index* selector uses numbers assigned at the time of creation of objects to identify the object in a window. Use this selector if the location of objects with the same description may change within a window. See "Configuring a Standard or Custom Class" on page 22 for more information.

Mapping a Custom Object to a Standard Class

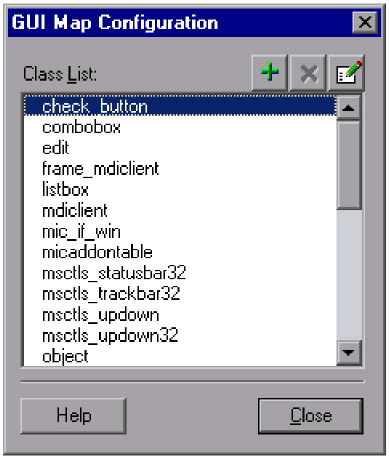
A custom object is any GUI object not belonging to one of the standard classes used by WinRunner. WinRunner learns such objects under the generic "object" class. WinRunner records operations on custom objects using **obj_mouse_** statements.

Using the GUI Map Configuration dialog box, you can teach WinRunner a custom object and map it to a standard class. For example, if your application has a custom button that WinRunner cannot identify, clicking this button is recorded as **obj_mouse_click**. You can teach WinRunner the "SampleCustomButtonClass" custom class and map it to the standard push_button class. Then, when you click the button, the operation is recorded as **button_press**.

Note that a custom object should be mapped only to a standard class with comparable behavior. For example, you cannot map a custom push button to the edit class.

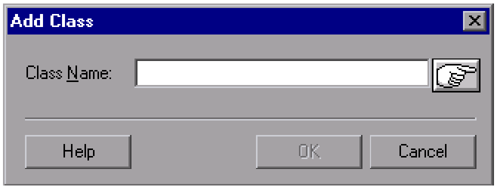
To map a custom object to a standard class:

1 Choose **Tools** > **GUI Map Configuration** to open the GUI Map Configuration dialog box. The Class List displays all standard and custom classes identified by WinRunner.



check_button
combobox
edit
frame_mdiclient
listbox
mdiclient
mic_if_win
micaddontable
msctls_statusbar32
msctls_trackbar32
msctls_updown
msctls_updown32
object

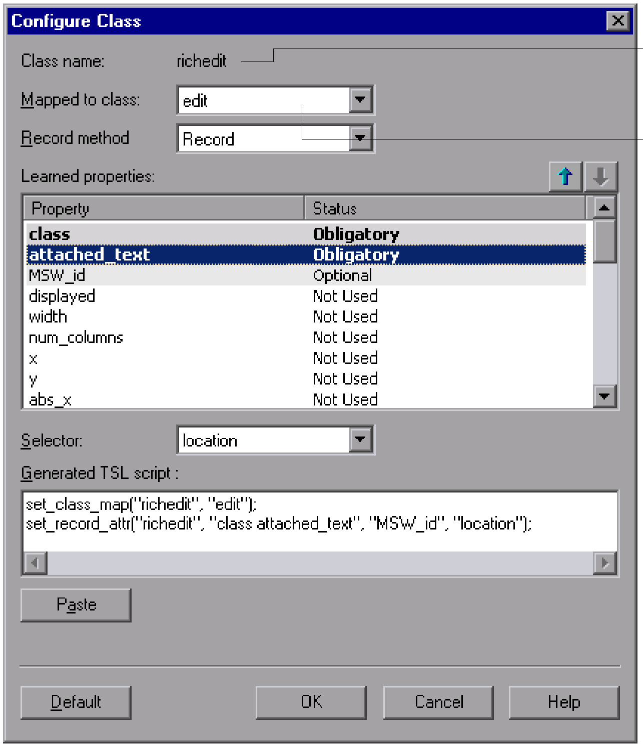
2 Click the **Add** button to open the Add Class dialog box.



3 Click the pointing hand and then click the object whose class you want to add. The name of the custom object appears in the Class Name box. Note that this name is the value of the object's MSW_class property.

4 Click **OK** to close the dialog box. The new class appears highlighted at the bottom of the Class List in the GUI Map Configuration dialog box, preceded by the letter "U" (user-defined).

5 Click the **Configure** button to open the Configure Class dialog box.



Class name: richedit

Mapped to class: edit

Record method: Record

Learned properties:

Property	Status
class	Obligatory
attached_text	Obligatory
MSW_id	Optional
displayed	Not Used
width	Not Used
num_columns	Not Used
x	Not Used
y	Not Used
abs_x	Not Used

Selector: location

Generated TSL script :

```
set_class_map("richedit", "edit");
set_record_attr("richedit", "class attached_text", "MSW_id", "location");
```

The custom class you are mapping

The list of standard classes

The **Mapped to class** box displays the object class. The object class is the class that WinRunner uses by default for all custom objects.

6 From the **Mapped to class** list, choose the standard class to which you want to map the custom class. Remember that you should map the custom class only to a standard class of comparable behavior.

Once you choose a standard class, the dialog box displays the GUI map configuration for that class.

You can also modify the GUI map configuration of the custom class (the properties learned, the selector, or the record method). For details, see "Configuring a Standard or Custom Class" on page 22.

7 Click **OK** to complete the configuration.

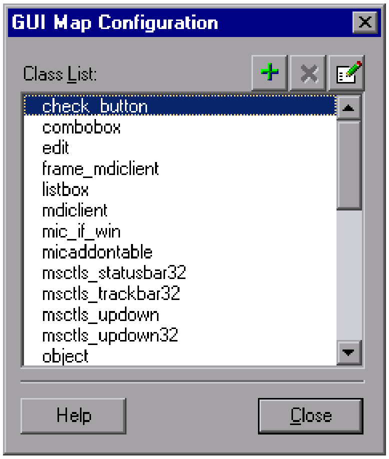
Note that the configuration is valid only for the current testing session. To make the configuration permanent, you should paste the TSL statements into a startup test script. See “Creating a Permanent GUI Map Configuration” on page 27 for more information.

Configuring a Standard or Custom Class

For any of the standard or custom classes, you can modify the properties learned, the selector, and/or the recording method.

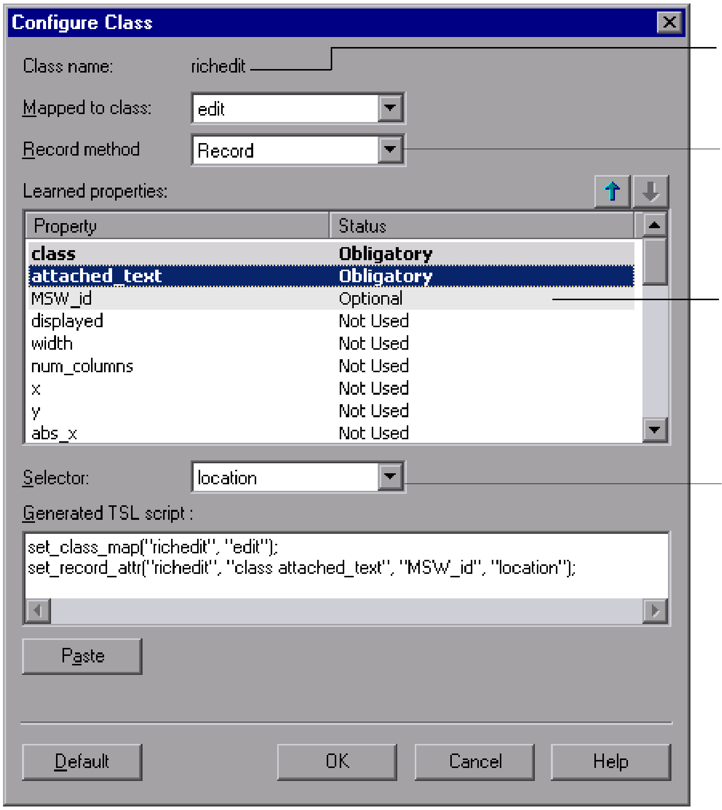
To configure a standard or custom class:

1 Choose **Tools** > **GUI Map Configuration** to open the GUI Map Configuration dialog box.



The Class List contains all standard classes, as well as any custom classes you add.

2 Choose the class you want to configure and click the **Configure** button. The Configure Class dialog box opens.



Configure Class

Class name: richedit

Mapped to class: edit

Record method: Record

Learned properties:

Property	Status
class	Obligatory
attached_text	Obligatory
MSW_id	Optional
displayed	Not Used
width	Not Used
num_columns	Not Used
x	Not Used
y	Not Used
abs_x	Not Used

Selector: location

Generated TSL script :

```
set_class_map("richedit", "edit");
set_record_attr("richedit", "class attached_text", "MSW_id", "location");
```

Class you want to configure

Record method for the class

Properties learned for the class

Selector for the class

The **Class name** field at the top of the dialog box displays the name of the class to configure.

3 Modify the learned properties, the selector, or the recording method as desired. See “Configuring Learned Properties” on page 24, “Configuring the Selector” on page 26, and “Configuring the Recording Method” on page 27 for details.

4 Click **OK**.

Note that the configuration is valid only for the current testing session. To make the configuration permanent, you should paste the TSL statements into a startup test script. See "Creating a Permanent GUI Map Configuration" on page 27 for more information.

5. Click **OK** in the GUI Map Configuration dialog box.

Configuring Learned Properties

The **Learned properties** area of the Configure Class dialog box allows you to configure which properties are recorded and learned for a class. You do this by changing in the status of the properties in order to signify whether they are obligatory, optional, or not used.

- Obligatory properties are always learned (provided that they are valid for the specific object).
- Optional properties are used only if the obligatory properties do not provide a unique identification for an object. WinRunner selects the minimum number of properties needed to identify the object, beginning with the first property in the list.
- The remaining properties are not used.

When the dialog box is displayed, the **Learned property** list displays the properties learned for the class appearing in the Class Name field.

The order in which the properties appear is important. Obligatory properties always appear at the top of the list, then optional properties, and finally not used properties. If WinRunner cannot identify the object using the obligatory properties, it refers to the optional properties in the order they appear in the list. You can adjust the position of a property by selecting it and clicking the **Up** or **Down** buttons.

To modify the property configuration:

1. Click the **Status** cell of the property whose status you want to change.
2. Select either Obligatory, Optional or Not Used from the list.
3. Click **OK** to save the changes.

Note that not all properties apply to all classes. The following table lists each property and the classes to which it can be applied.

Property	**Classes**
abs_x	All classes
abs_y	All classes
active	All classes
attached_text	combobox, edit, listbox, scrollbar
class	All classes
displayed	All classes
enabled	All classes
focused	All classes
handle	All classes
height	All classes
label	check_button, push_button, radio_button, static_text, window
maximizable	calendar, window
minimizable	calendar, window
MSW_class	All classes
MSW_id	All classes, except window
nchildren	All classes
obj_col_name	edit
owner	mdiclient, window
pb_name	check_button, combobox, edit, list, push_button, radio_button, scroll, window (object)
regexp_label	All classes with labels
regexp_ MSWclass	All classes
text	All classes

Property	Classes
value	calendar, check_button, combobox, edit, listbox, radio_button, scrollbar, static_text
vb_name	All classes
virtual	list, push_button, radio_button, table, object (virtual objects only)
width	All classes
x	All classes
y	All classes

Configuring the Selector

In cases where both obligatory and optional properties cannot uniquely identify an object, WinRunner applies one of two selectors: *location* or *index*.

A location selector performs the selection process based on the position of objects within the window: from top to bottom and from left to right. An index selector performs a selection according to a unique number assigned to an object by the application developer. For an example of how selectors are used, see “Understanding the Default GUI Map Configuration” on page 18.

By default, WinRunner uses a location selector for all classes. To change the selector, click the appropriate radio button.

Configuring the Recording Method

By setting the recording method you can determine how WinRunner records operations on objects belonging to the same class. Three recording methods are available:

- *Record* instructs WinRunner to record all operations performed on a GUI object. This is the default record method for all classes. (The only exception is the static class (static text), for which the default is *Pass Up*.)
- *Pass Up* instructs WinRunner to record an operation performed on this class as an operation performed on the element containing the object. Usually this element is a window, and the operation is recorded as **win_mouse_click**.
- *As Object* instructs WinRunner to record all operations performed on a GUI object as though its class were "object" class.
- *Ignore* instructs WinRunner to disregard all operations performed on the class.

To modify the recording method, click the appropriate radio button.

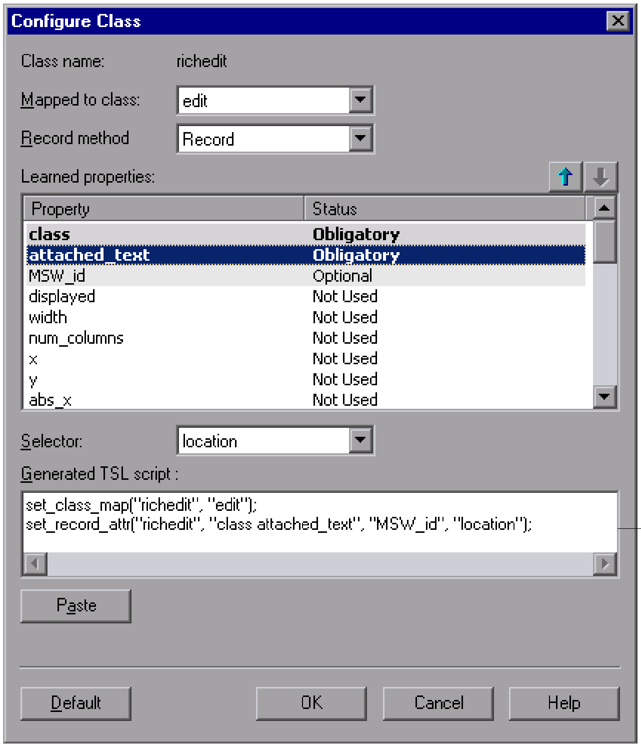
Creating a Permanent GUI Map Configuration

By generating TSL statements describing the configuration you set and inserting them into a startup test, you can ensure that WinRunner always uses the correct GUI map configuration for your standard and custom object classes.

To create a permanent GUI map configuration for a class:

1. Choose **Tools** > **GUI Map Configuration** to open the GUI Map Configuration dialog box.
2. Choose a class and click the **Configure** button. The Configure Class dialog box opens.

3 Set the desired configuration for the class. Note that in the bottom pane of the dialog box, WinRunner automatically generates the appropriate TSL statements for the configuration.



Class name: richedit

Mapped to class: edit

Record method: Record

Learned properties:

Property	Status
class	**Obligatory**
attached_text	**Obligatory**
MSW_id	Optional
displayed	Not Used
width	Not Used
num_columns	Not Used
x	Not Used
y	Not Used
abs_x	Not Used

Selector: location

Generated TSL script :

```
set_class_map("richedit", "edit");
set_record_attr("richedit", "class attached_text", "MSW_id", "location");
```

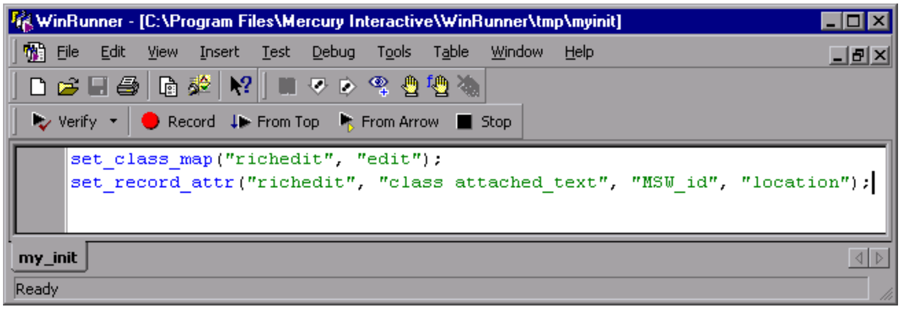
TSL statements describing the GUI map configuration

4 Paste the TSL statements into a startup test using the **Paste** button.

For example, assume that in the WinRunner configuration file *wrun.ini* (located in your Windows folder), your startup test is defined as follows:

```
[WrEnv]
XR_TSL_INIT = C:\tests\my_init
```

You would open the my_init test in the WinRunner window and paste in the generated TSL lines.



WinRunner - [C:\Program Files\Mercury Interactive\WinRunner\tmp\myinit]
File Edit View Insert Test Debug Tools Table Window Help
Verify
Record
From Top
From Arrow
Stop
my_init
Ready

```
set_class_map("richedit", "edit");
set_record_attr("richedit", "class attached_text", "MSW_id", "location");
```

For more information on startup tests, see Chapter 23, “Initializing Special Configurations.” For more information on the TSL functions defining a custom GUI map configuration (**set_class_map**, **set_record_attr**, and **set_record_method**), refer to the *TSL Reference*.

Deleting a Custom Class

You can delete only custom object classes. The standard classes used by WinRunner cannot be deleted.

To delete a custom class:

1. Choose **Tools** > **GUI Map Configuration** to open the GUI Map Configuration dialog box.
2. Choose the class you want to delete from the Class list.
3. Click the **Delete** button.

Understanding WinRunner Object Classes

WinRunner categorizes GUI objects according to the following classes according to a number of object classes. When viewing the props of any object, the class property indicates the object property class of the object. WinRunner supports the following object classes.:

Class	**Description**
calendar	A standard calendar object that belongs to the *CDateTimeCtrl* or *CMonthCalCtrl* MSW_class.
check_button	A check box.
edit	An edit field.
frame_mdiclient	Enables WinRunner to treat a window as an mdiclient object.
list	A list box. This can be a regular list or a combo box.
menu_item	A menu item.
mdiclient	An mdiclient object.
mic_if_win	Enables WinRunner to defer all record and run operations on any object within this window to the mic_if library. Refer to the *WinRunner Customization Guide* for more information.
object	Any object not included in one of the classes described in this table.
push_button	A push (command) button.
radio_button	A radio (option) button.
scroll	A scroll bar or slider.
spin	A spin object.
static_text	Display-only text not part of any GUI object.
status bar	A status bar on a window.
tab	A tab item.
toolbar	A toolbar object.
window	Any application window, dialog box, or form, including MDI windows.

Understanding Object Properties

The following tables list all properties used by WinRunner in Context Sensitive testing.

Property	Description
abs_x	The x-coordinate of the top left corner of an object, relative to the origin (upper left corner) of the screen display.
abs_y	The y-coordinate of the top left corner of an object, relative to the origin (upper left corner) of the screen display.
active	A Boolean value indicating whether this is the top-level window associated with the input focus.
attached_text	The static text located near the object.
class	The WinRunner class of the GUI object. For more information, see "Understanding WinRunner Objects Classes" on page 30.
class_index	An index number that identifies an object, relative to the position of other objects from the same class in the window (Java add-in only).
count	The number of menu items contained in a menu.
displayed	A Boolean value indicating whether the object is displayed: 1 if visible on screen, 0 if not.
enabled	A Boolean value indicating whether the object can be selected or activated: 1 if enabled, 0 if not.
focused	A Boolean value indicating whether keyboard input will be directed to this object: 1 if object has keyboard focus, 0 if not.
handle	A run-time pointer to the object: the HWND handle.
height	Height of object in pixels.
label	The text that appears on the object, such as a button label.
maximizable	A Boolean value indicating whether a window can be maximized: 1 if the window can be maximized, 0 if not.
minimizable	A Boolean value indicating whether a window can be minimized: 1 if the window can be minimized, 0 if not.

Property	Description
module_name	The name of an executable file which created the specified window.
MSW_class	The Microsoft Windows class.
MSW_id	The Microsoft Windows ID.
nchildren	The number of children the object has: the total number of descendants of the object.
num_columns	A table object in Terminal Emulator applications only.
num_rows	A table object in Terminal Emulator applications only.
obj_col_name	A concatenation of the DataWindow and column names. For edit field objects in WinRunner with PowerBuilder add-in support, indicates the name of the column.
owner	(For windows), the application (executable) name to which the window belongs.
parent	The logical name of the parent of the object.
pb_name	A text string assigned to PowerBuilder objects by the developer. (The property applies only to WinRunner with PowerBuilder add-in support.)
position	The position (top to bottom) of a menu item within the menu (the first item is at position 0).
regexp_label	The text string and regular expression that enables WinRunner to identify an object with a varying label.
regexp_MSWclass	The Microsoft Windows class combined with a regular expression. Enables WinRunner to identify objects with a varying MSW_class.
submenu	A Boolean value indicating whether a menu item has a submenu: 1 if menu has submenu, 0 if not.
sysmenu	A Boolean value indicating whether a menu item is part of a system menu.
TOOLKIT_class	The value of the specified toolkit class. The value of this property is the same as the value of the MSW_class in Windows, or the X_class in Motif.

Property	Description
text	The visible text in an object or window.
value	Different for each class: Radio and check buttons: 1 if the button is checked, 0 if not. Menu items: 1 if the menu is checked, 0 if not. List objects: indicates the text string of the selected item. Edit/Static objects: indicates the text field contents. Scroll objects: indicates the scroll position. All other classes: the value property is a null string.
vb_name	A text string assigned to Visual Basic objects by the developer (the *name* property). (The property applies only to WinRunner with Visual Basic add-in support.)
width	Width of object in pixels.
x	The x-coordinate of the top left corner of an object, relative to the window origin.
y	The y-coordinate of the top left corner of an object, relative to the window origin.

Understanding Default Learned Properties

The following table lists the default properties learned for each class. (The default properties apply to all methods of learning: the RapidTest Script wizard, the GUI Map Editor, and recording.)

Class	Obligatory Properties	Optional Properties	Selector
All buttons	class, label	MSW_id	location
list, edit, scroll, combobox	class, attached_text	MSW_id	location
frame_mdiclient	class, regexp_MSWclass, regexp_label	label, MSW_class	location
menu_item	class, label, sysmenu	position	location
object	class, regexp_MSWclass, label	attached_text, MSW_id, MSW_class	location
mdiclient	class, label	regexp_MSWclass, MSW_class	
static_text	class, MSW_id	label	location
window	class, regexp_MSWclass, label	attached_text, MSW_id, MSW_class	location

Properties for Visual Basic Objects

The label and vb_name properties are obligatory properties: they are learned for all classes of Visual Basic objects. For more information on testing Visual Basic objects, refer to Chapter 11, “Working with ActiveX and Visual Basic Controls” in the *Mercury WinRunner Basic Features User’s Guide*.

Note: To test Visual Basic applications, you must install Visual Basic support. For more information, refer to the *WinRunner Installation Guide*.

Properties for PowerBuilder Objects

The following table lists the standard object classes and the properties learned for each PowerBuilder object. For more information on testing PowerBuilder objects, refer to Chapter 12, “Checking PowerBuilder Applications” in the *Mercury WinRunner Basic Features User’s Guide*.

Class	Obligatory Properties	Optional Properties	Selector
all buttons	class, pb_name	label, MSW_id	location
list, scroll, combobox	class, pb_name	attached_text, MSW_id	location
edit	class, pb_name, obj_col_name	attached_text, MSW_id	location
object	class, pb_name	label, attached_text, MSW_id, MSW_class	location
window	class, pb_name	label, MSW_id	location

Note: In order to test PowerBuilder applications, you must install PowerBuilder support. For more information, refer to the *WinRunner Installation Guide*.

3

Learning Virtual Objects

You can teach WinRunner to recognize any bitmap in a window as a GUI object by defining the bitmap as a *virtual object*.

This chapter describes:

- About Learning Virtual Objects
- Defining a Virtual Object
- Understanding a Virtual Object's Physical Description

About Learning Virtual Objects

Your application may contain bitmaps that look and behave like GUI objects. WinRunner records operations on these bitmaps using **win_mouse_click** statements. By defining a bitmap as a *virtual object,* you can instruct WinRunner to treat it like a GUI object such as a push button, when you record and run tests. This makes your test scripts easier to read and understand.

For example, suppose you record a test on the Windows NT Calculator application in which you click buttons to perform a calculation. Since WinRunner cannot recognize the calculator buttons as GUI objects, by default it creates a test script similar to the following:

```
set_window("Calculator");
win_mouse_click ("Calculator", 87, 175);
win_mouse_click ("Calculator", 204, 200);
win_mouse_click ("Calculator", 121, 163);
win_mouse_click ("Calculator", 242, 201);
```

This test script is difficult to understand. If, instead, you define the calculator buttons as virtual objects and associate them with the push button class, WinRunner records a script similar to the following:

```
set_window ("Calculator");
button_press("seven");
button_press("plus");
button_press("four");
button_press("equal");
```

You can create virtual push buttons, radio buttons, check buttons, lists, or tables, according to the bitmap's behavior in your application. If none of these is suitable, you can map a virtual object to the general object class.

You define a bitmap as a virtual object using the Virtual Object wizard. The wizard prompts you to select the standard class with which you want to associate the new object. Then you use a crosshairs pointer to define the area of the object. Finally, you choose a logical name for the object. WinRunner adds the virtual object's logical name and physical description to the GUI map.

Defining a Virtual Object

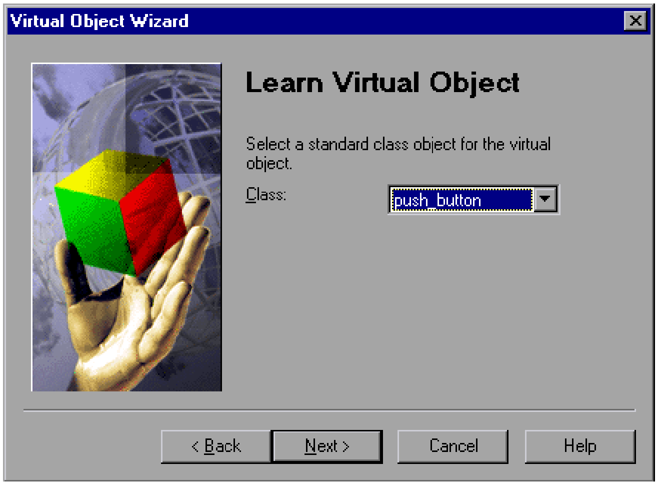
Using the Virtual Object wizard, you can assign a bitmap to a standard object class, define the coordinates of that object, and assign it a logical name.

To define a virtual object using the Virtual Object wizard:

1 Choose **Tools** > **Virtual Object Wizard**. The Virtual Object wizard opens.

Click **Next**.

2 In the Class list, select a class for the new virtual object.



Virtual Object Wizard

Learn Virtual Object

Select a standard class object for the virtual object.

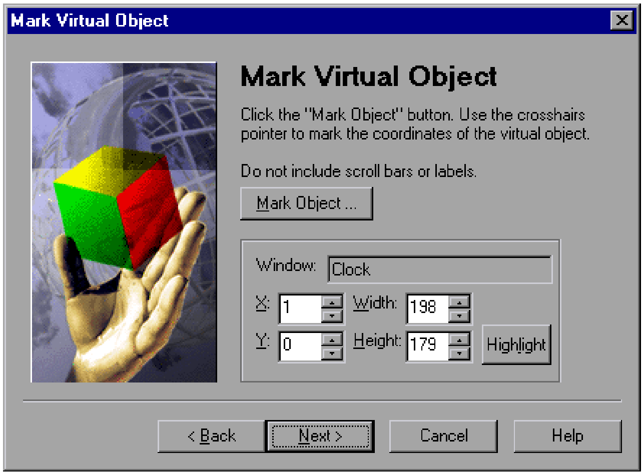
Class: push_button

If you select the **list** class, select the number of visible rows that are displayed in the window. For a **table** class, select the number of visible rows and columns. Click **Next**.

3 Click **Mark Object**. Use the crosshairs pointer to select the area of the virtual object. You can use the arrow keys to make precise adjustments to the area you define with the crosshairs.

Note: The virtual object should not overlap GUI objects in your application (except for those belonging to the generic Object class, or to a class configured to be recorded as Object). If a virtual object overlaps a GUI object, WinRunner may not record or execute tests properly on the GUI object.

Press ENTER or click the right mouse button to display the virtual object's coordinates in the wizard. If the object marked is visible on the screen, you can click the **Highlight** button to view it.



Mark Virtual Object

Click the "Mark Object" button. Use the crosshairs pointer to mark the coordinates of the virtual object.

Do not include scroll bars or labels.

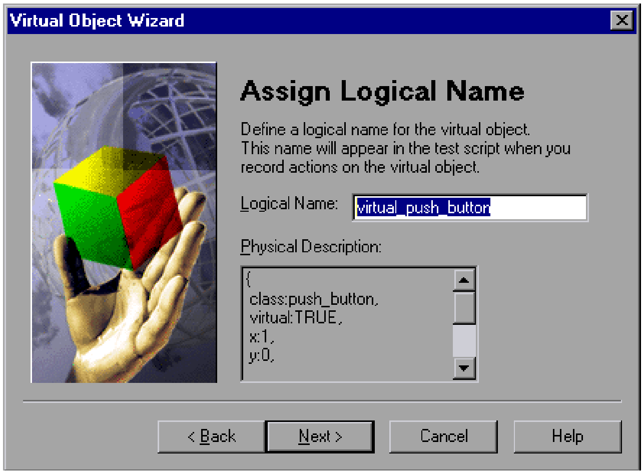
Window: Clock

X: 1 Width: 198

Y: 0 Height: 179

Click **Next**.

4 Assign a logical name to the virtual object. This is the name that appears in the test script when you record on the virtual object. If the object contains text that WinRunner can read, the wizard suggests using this text for the logical name. Otherwise, WinRunner suggests *virtual_object*, *virtual_push_button*, *virtual_list*, etc.



Assign Logical Name

Define a logical name for the virtual object.
This name will appear in the test script when you record actions on the virtual object.

Logical Name: virtual_push_button

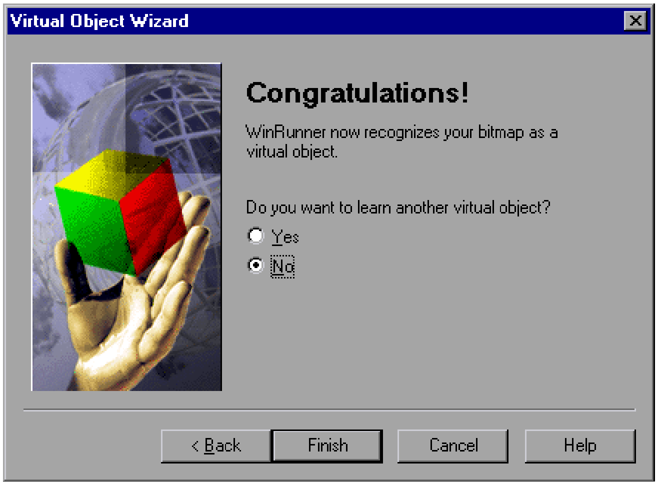
Physical Description:

```
{
class:push_button,
virtual:TRUE,
x:1,
y:0,
```

You can accept the wizard's suggestion or type in a different name. WinRunner checks that there are no other objects in the GUI map with the same name before confirming your choice. Click **Next**.

5 Finish learning the virtual object:

- If you want to learn another virtual object, choose **Yes** and click **Next**.
- To close the wizard, choose **No** and click **Finish**.



Congratulations!

WinRunner now recognizes your bitmap as a virtual object.

Do you want to learn another virtual object?

○ Yes
● No

When you exit the wizard, WinRunner adds the object's logical name and physical description to the GUI map. The next time that you record operations on the virtual object, WinRunner generates TSL statements instead of **win_mouse_click** statements.

Understanding a Virtual Object’s Physical Description

When you create a virtual object, WinRunner adds its physical description to the GUI map. The physical description of a virtual object does not contain the *label* property found in the physical description of “real” GUI objects. Instead it contains a special property, *virtual*. Its function is to identify virtual objects, and its value is always TRUE.

Since WinRunner identifies a virtual object according to its size and its position within a window, the x, y, width, and height properties are always found in a virtual object’s physical description.

For example, the physical description of a *virtual_push_button* includes the following properties:

```
{
 class: push_button,
 virtual: TRUE,
 x: 82,
 y: 121,
 width: 48,
 height: 28,
}
```

If these properties are changed or deleted, WinRunner cannot recognize the virtual object. If you move or resize an object, you must use the wizard to create a new virtual object.

Part II

Creating Tests—Advanced

4

Defining and Using Recovery Scenarios

You can instruct WinRunner to recover from unexpected events and errors that occur in your testing environment during a test run.

This chapter describes:

About Defining and Using Recovery Scenarios

Unexpected events, errors, and application crashes during a test run can disrupt your test and distort test results. This is a problem particularly when running batch tests unattended: the batch test is suspended until you perform the action needed to recover.

The Recovery Manager provides a wizard that guides you through the process of defining a *recovery scenario*: an unexpected event and the operation(s) necessary to recover the test run. For example, you can instruct WinRunner to detect a “Printer out of paper” message and recover the test run by clicking the **OK** button to close the message, and continue the test from the point at which the test was interrupted.

There are two types of recovery scenarios:

- **Simple:** Enables you to define a (non-crash) exception event and the single operation that will terminate the event, so that the test can continue.
- **Compound:** an exception or crash event and the operation(s) required to continue or restart the test and the associated applications.

A recovery scenario has two main components:

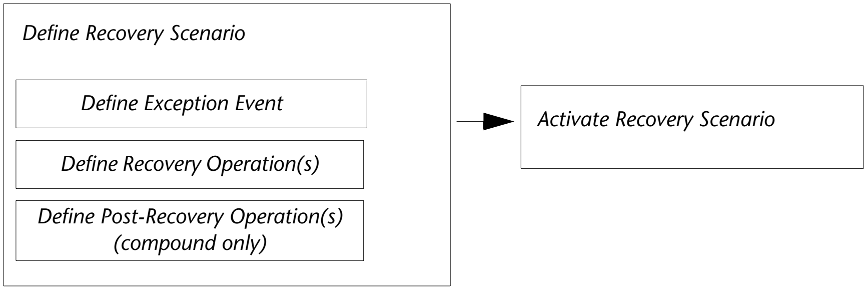
- **Exception Event**: The event that interrupts your test run.
- **Recovery Operation(s)**: The operation(s) that terminate the interruption.

Compound recovery scenarios also include **Post-Recovery Operation(s)**, which provide instructions on how WinRunner should proceed once the recovery operations have been performed, including any functions WinRunner should run before continuing, and from which point in the test or batch WinRunner should continue, if at all. For example, you may need to run a function that reopens certain applications and sets them to the proper state, and then restart the test that was interrupted from the beginning.

The functions that you specify for recovery and post-recovery operations can come from any regular compiled module, or they can come from the recovery compiled module. The *recovery compiled module* is a special compiled module that is always loaded when WinRunner opens so that the functions it contains can be accessed whenever WinRunner performs a recovery scenario.

To instruct WinRunner to perform a recovery scenario during a test run, you must activate it.

The following diagram summarizes the steps involved in creating a recovery scenario:


Define Recovery Scenario
Define Exception Event
Define Recovery Operation(s)
Define Post-Recovery Operation(s) (compound only)
Activate Recovery Scenario

Recovery scenarios apply only to Windows events. You can also define Web exceptions and handler functions. For more information, see Chapter 5, "Handling Web Exceptions."

Defining Simple Recovery Scenarios

A simple recovery scenario defines a non-crash exception event and the single operation that will terminate the event, so that the test can continue.

You can define and modify simple recovery scenarios from the **Simple** tab of the Recovery Manager. The Recovery wizard guides you through the process of creating or modifying your scenario.

You can also define simple recovery scenarios using TSL statements. For more information, see "Working with Recovery Scenarios in Your Test Script" on page 84.

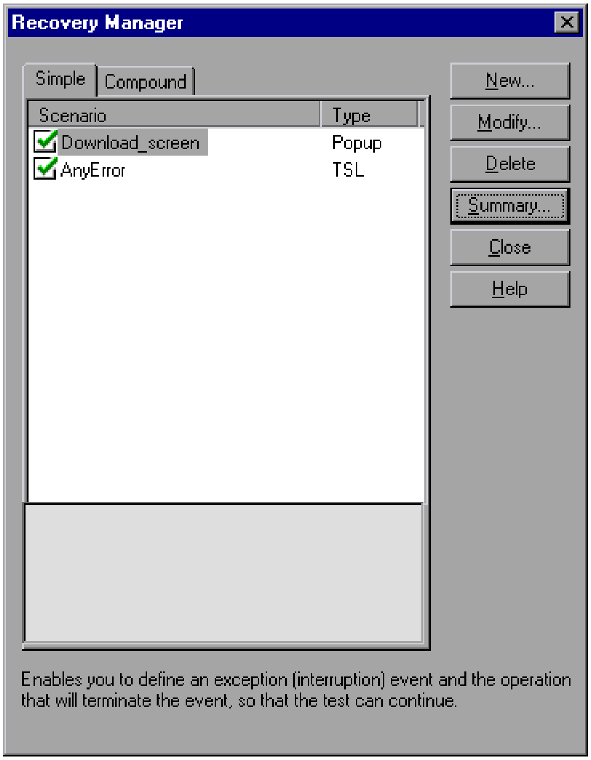
Notes:

The simple recovery scenario parallels what was formerly called exception handling. Exceptions created in the Exception Handler in WinRunner 7.01 or earlier are displayed in the **Simple** tab of the Recovery Manager.

The first time you use the Recovery Manager to add, modify, or delete a recovery scenario, WinRunner prompts you to select a new recovery scenarios file. For more information, see "Working with Recovery Scenarios Files" on page 80.

To create a simple recovery scenario

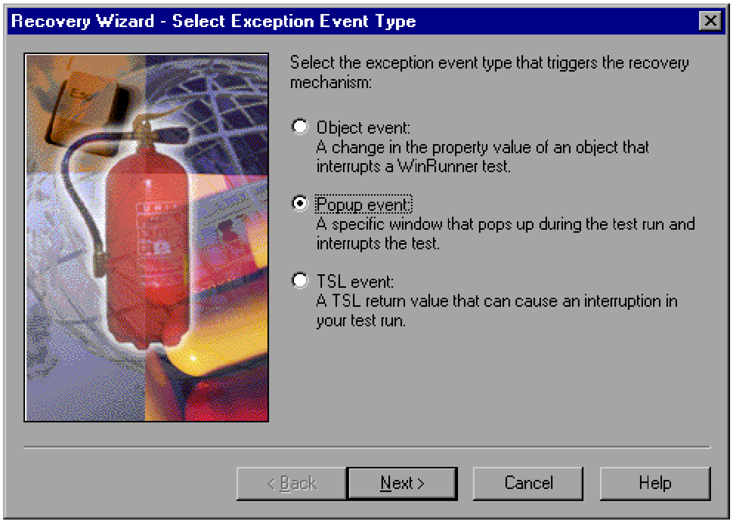
1 Choose **Tools** > **Recovery Manager**. The Recovery Manager opens.



Scenario	Type
☑ Download_screen	Popup
☑ AnyError	TSL

Enables you to define an exception (interruption) event and the operation that will terminate the event, so that the test can continue.

2 Click **New**. The Recovery wizard opens to the Select Exception Event Type screen.



Select the exception event type that triggers the recovery mechanism:

○ Object event:
A change in the property value of an object that interrupts a WinRunner test.

● Popup event:
A specific window that pops up during the test run and interrupts the test.

○ TSL event:
A TSL return value that can cause an interruption in your test run.

3 Select the exception event type that triggers the recovery mechanism.

➤ **Object event:** a change in the property value of an object that causes an interruption in the WinRunner test.

For example, suppose that your application uses a green button to indicate that an electrical network is closed; the same button may turn red when the network is broken. Your test cannot continue while the network is broken.

➤ **Popup event:** a window that pops up during the test run and interrupts the test.

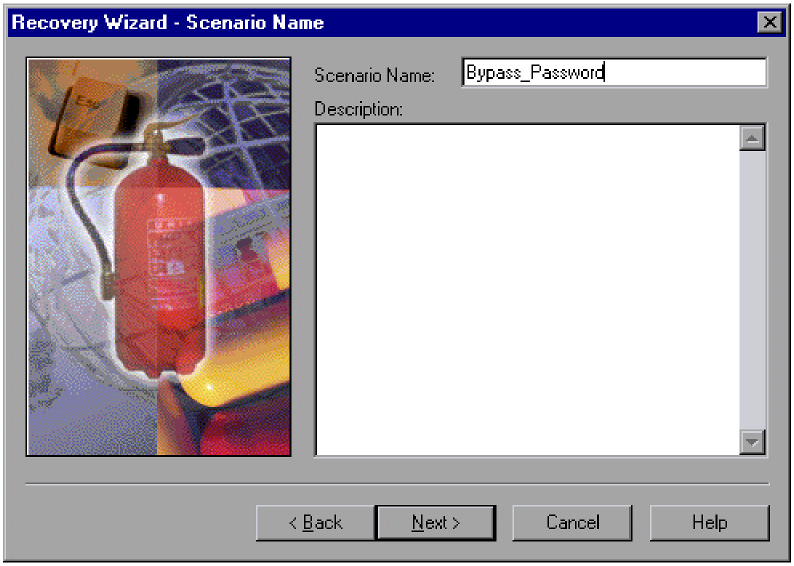
For example, suppose part of your test includes clicking on a Print button to send a generated graph to the printer, and a message box opens indicating that the printer is out of paper. Your test cannot continue until you close the message box.

➤ **TSL event:** a TSL return value that can cause an interruption in the test run.

For example, suppose a **set_window** statement returns an error. You could use a recovery scenario to close, initialize, and reopen the window.

Click **Next**.

4 The Scenario Name screen opens.



Recovery Wizard - Scenario Name

Scenario Name: Bypass_Password

Description:

Enter a name containing only alphanumeric characters and underscores (no spaces or special characters) and a description for your recovery scenario.

Click **Next**.

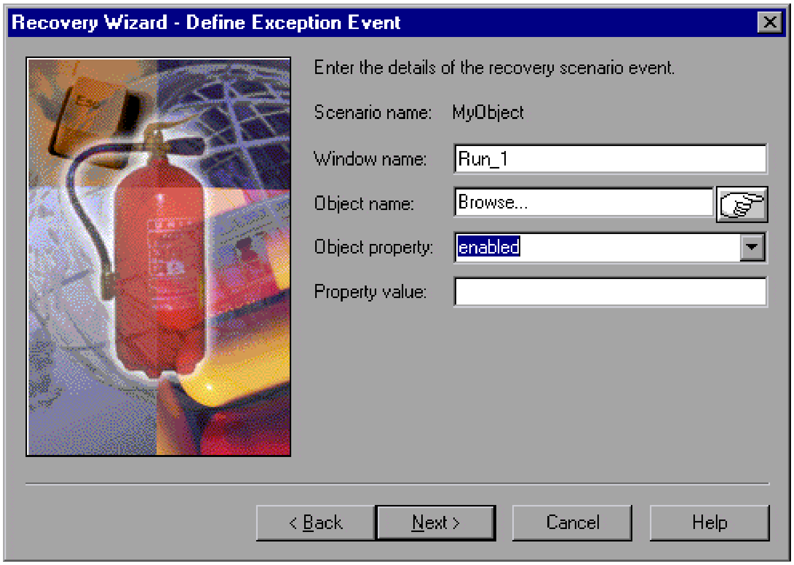
5 The Define Exception Event screen opens. The options in the screen vary based on the type of event you selected in step 3.

For information on defining object events, see page 51.

For information on defining pop-up events, see page 53.

For information on defining TSL events, see page 54.

If you chose an object event in step 3, enter the following information:



Recovery Wizard - Define Exception Event

Enter the details of the recovery scenario event.

Scenario name: MyObject

Window name: Run_1

Object name: Browse...

Object property: enabled

Property value:

- **Window name:** Indicates the name of the window containing the object that causes the exception. Enter the logical name of the window, or use the pointing hand next to the Object name box to click on the object you want to define for the object exception and WinRunner will automatically fill in the Window name and Object name.

  If you want to define a window as the exception object, click on the window's title bar, or enter the window's logical name and leave the Object name box empty.

- **Object name:** Indicates the name of the object that causes the exception. Enter the logical name of the object, or use the pointing hand next to the Object name box to specify the object you want to define for the object exception and WinRunner will automatically fill in the Window name and Object name.

Note: The object you define must be saved in the GUI Map. If the object is not already saved in the GUI Map and you use the pointing hand to identify the object, WinRunner automatically adds it to the active GUI Map. If you type the object name manually, you must also add the object to the GUI Map. For more information on the GUI Map, refer to Chapter 4, “Understanding Basic GUI Map Concepts” in the *Mercury WinRunner Basic Features User’s Guide*.

- **Object property:** The object property whose value you want to check. Select the object property in which an exception may occur. For example, if want to detect when a button changes from enabled to disabled, select the enabled property.

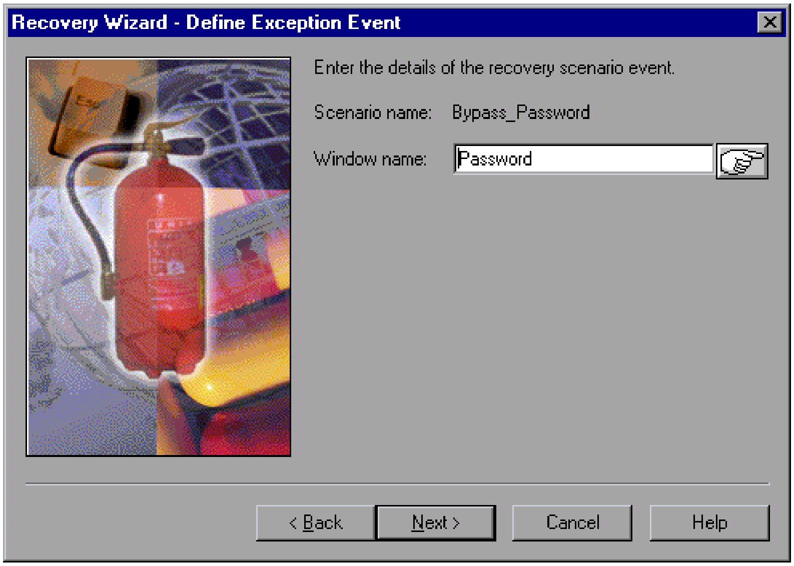
Note: You cannot specify a property that is part of the object's physical description.

- **Property value:** The value that indicates that an exception has occurred. For example, if you want WinRunner to activate the recovery scenario when the button changes from enabled to disabled, type 0 in the field.

Tip: Leave the property value empty to detect any change in the property value.

Click **Next** and proceed to step 6 on page 55.

If you chose a pop-up event in step 3, enter the following information:



Enter the details of the recovery scenario event.

Scenario name: Bypass_Password

Window name: Password

- **Window name:** Indicates the name of the pop-up window that causes the exception. Enter the logical name of the window, or use the pointing hand to specify the window you want to define as a pop-up exception.

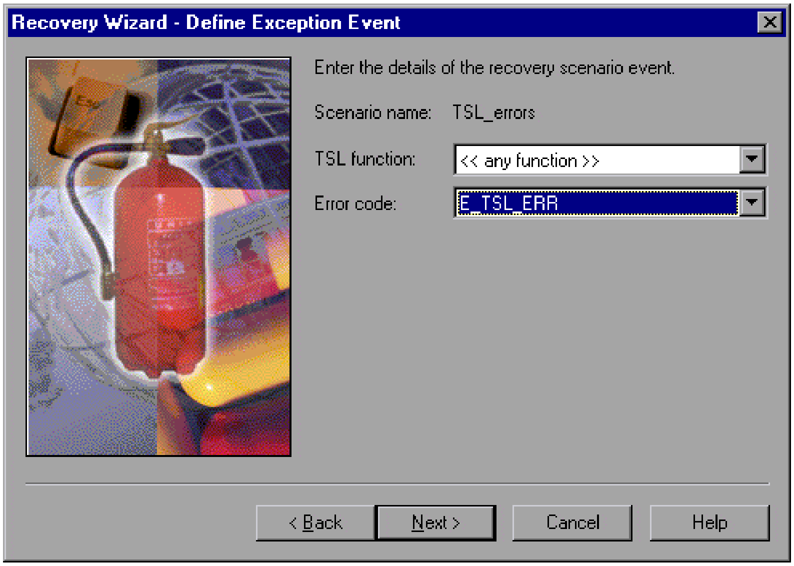
If the window is not already saved in the GUI Map and you use the pointing hand to identify the window, WinRunner automatically adds it to the active GUI Map. If the window is not already saved in the GUI Map and you type the name manually, WinRunner identifies the pop-up exception when a pop-up window opens with a title bar matching the name you entered.

Note: If you want to employ the **Click button** recovery operation, then the pop-up window you define must be saved in the GUI Map. If you type the window name manually, you must also add the window to the GUI Map. For more information about recovery operations, see page 55.

Tip: If the pop-up window that causes the exception has a window name that is generated dynamically, use the pointing hand to add the window to the GUI Map and then modify the definition of the window in the GUI Map using regular expressions.

Click **Next** and proceed to step 6 on page 55.

If you chose a TSL event in step 3, enter the following information:



Recovery Wizard - Define Exception Event

Enter the details of the recovery scenario event.

Scenario name: TSL_errors

TSL function: << any function >>

Error code: E_TSL_ERR

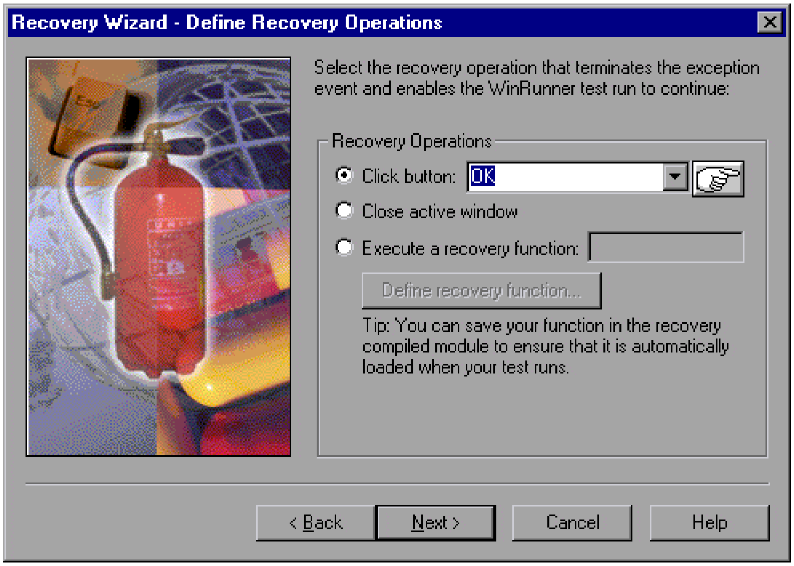
➤ **TSL function:** Select the TSL function for which you want to define the exception event. Select a TSL function from the list. WinRunner detects the exception only when the selected TSL function returns the code selected in the Error code box.

Tip: Select << **any function** >> to trigger the exception mechanism for any TSL function that returns the specified Error code.

- **Error code:** Select the TSL error code that triggers the exception mechanism. Select an error code from the list. WinRunner activates the recovery scenario when this return code is detected for the selected TSL function during a test run.

  Click **Next**.

6 The Define Recovery Operations screen opens.



Recovery Wizard - Define Recovery Operations

Select the recovery operation that terminates the exception event and enables the WinRunner test run to continue:

Recovery Operations

● Click button: OK

○ Close active window

○ Execute a recovery function:

Tip: You can save your function in the recovery compiled module to ensure that it is automatically loaded when your test runs.

Select one of the following recovery options:

- **Click button**: Specifies the logical name of the button to click on the pop-up window when the exception event occurs. Select one of the default button names, type the logical name of a button, or use the pointing hand to specify the button to click.

Notes:

This option is available only for pop-up exceptions.

The pop-up window defined for the recovery scenario must be defined in the GUI map. If the pop-up window is not defined in a loaded GUI map file when you define the pop-up recovery scenario, the recovery scenario will automatically be set as inactive. If you later load a GUI map containing the pop-up window, you can then activate the recovery scenario.

- **Close active window:** Instructs WinRunner to close the active (in focus) window when the exception event occurs.

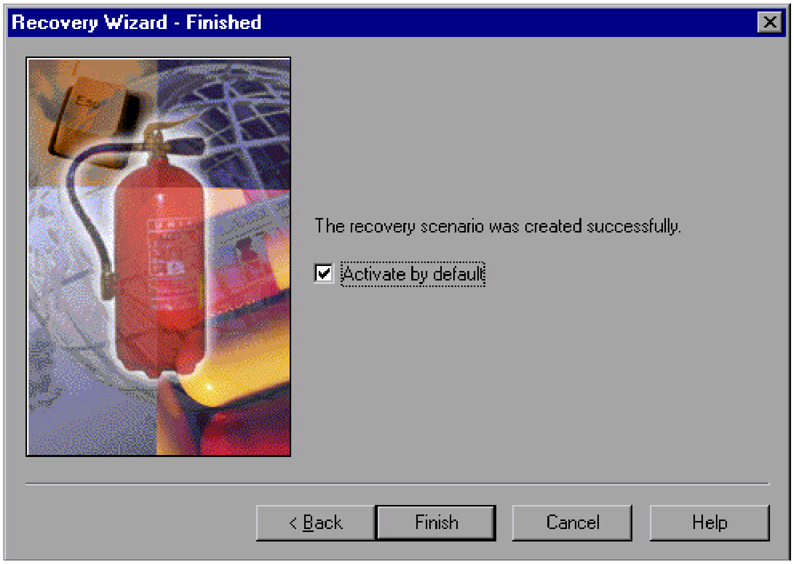
Note: WinRunner uses the (TSL) **win_close** mechanism to close the window. If the **win_close** function cannot close the window, the recovery scenario cannot close the window. In these situations, use the **Click button** or **Execute a recovery function** options instead.

- **Execute a recovery function**: Instructs WinRunner to run the specified function when the exception event occurs. You can specify an existing function or click **Define recovery function** to define a new function. For more information on defining recovery functions, see “Defining Recovery Scenario Functions” on page 73.

Note: The compiled module containing the function must be loaded when the test runs. Save your function in the recovery compiled module to ensure that it is always automatically loaded when WinRunner opens. If you do not select a function saved in the recovery compiled module, ensure that the compiled module containing your function is loaded whenever a recovery scenario using the function is activated.

Click **Next**.

7 The Finished screen opens.



Recovery Wizard - Finished

The recovery scenario was created successfully.

☑ Activate by default

Determine whether you want your recovery scenario to be activated by default when WinRunner opens:

- Select **Activate by default** to instruct WinRunner to automatically activate the recovery scenario by default when WinRunner opens, even if the scenario was set as inactive at the end of the previous WinRunner session.
- Clear **Activate by default** if you do not want WinRunner to automatically activate the recovery scenario by default when WinRunner opens. Note that if you clear this check box, your recovery scenario will not be activated unless you activate it manually by toggling the check box in the Recovery Manager dialog box.

For information on other ways to activate or deactivate a recovery scenario, see “Activating and Deactivating Recovery Scenarios” on page 78 and “Working with Recovery Scenarios in Your Test Script” on page 84.

Click **Finish**. The recovery scenario is added to the **Simple** tab of the Recovery Manager dialog box. If you selected Activate by default (and any required objects are found in the loaded GUI map file(s)), the recovery scenario is activated. Otherwise the recovery scenario remains inactive.

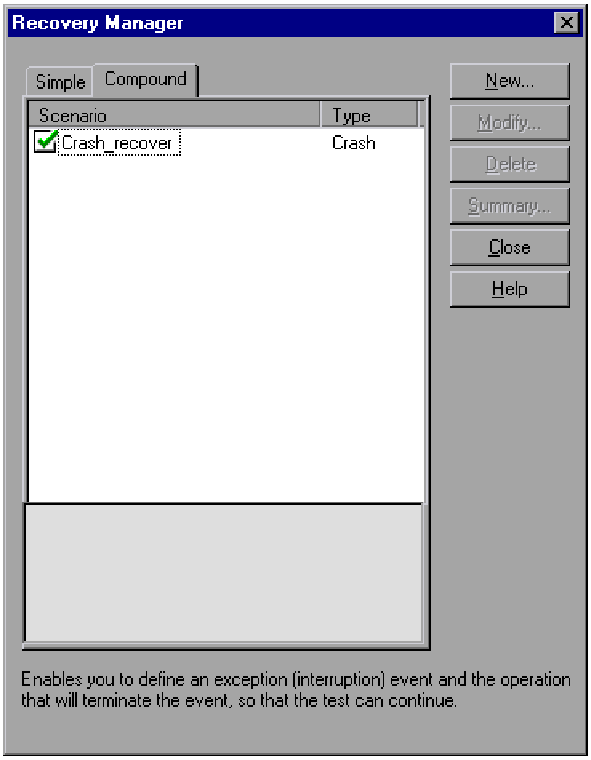
Defining Compound Recovery Scenarios

A compound recovery scenario defines a crash or exception event and the operation(s) required to continue or restart the test and the associated applications. You define and modify compound recovery scenarios from the **Compound** tab of the Recovery Manager. The Recovery wizard guides you through the process of creating and modifying your scenario.

To create a compound recovery scenario:

1 Choose **Tools** > **Recovery Manager**. The Recovery Manager opens.

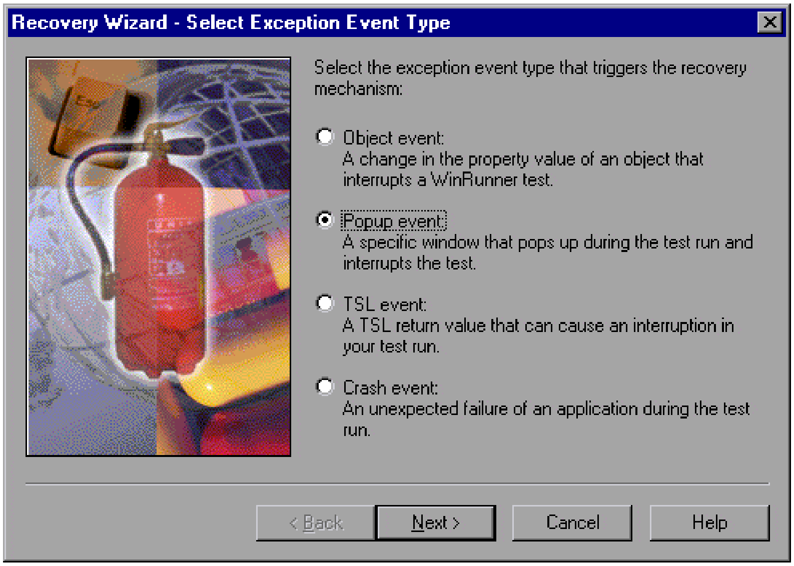
2 Click the **Compound** tab.



Scenario	Type
✓ Crash_recover	Crash

Enables you to define an exception (interruption) event and the operation that will terminate the event, so that the test can continue.

3 Click **New**. The Recovery wizard opens to the Select Exception Event Type screen.



Recovery Wizard - Select Exception Event Type

Select the exception event type that triggers the recovery mechanism:

○ Object event:
A change in the property value of an object that interrupts a WinRunner test.

● Popup event:
A specific window that pops up during the test run and interrupts the test.

○ TSL event:
A TSL return value that can cause an interruption in your test run.

○ Crash event:
An unexpected failure of an application during the test run.

4 Select the exception event type that triggers the recovery mechanism.

- **Object event:** a change in the property value of an object that causes an interruption in the WinRunner test.

  For example, suppose that your application uses a green button to indicate that an electrical network is closed; the same button may turn red when the network is broken. Your test cannot continue while the network is broken.

- **Popup event:** a window that pops up during the test run and interrupts the test.

  For example, suppose part of your test includes clicking on a Print button to send a generated graph to the printer, and a message box opens indicating that the printer is out of paper. Your test cannot continue until you close the message box.

- **TSL event:** a TSL return value that can cause an interruption in the test run.

- **Crash event:** an unexpected failure of an application during the test run.

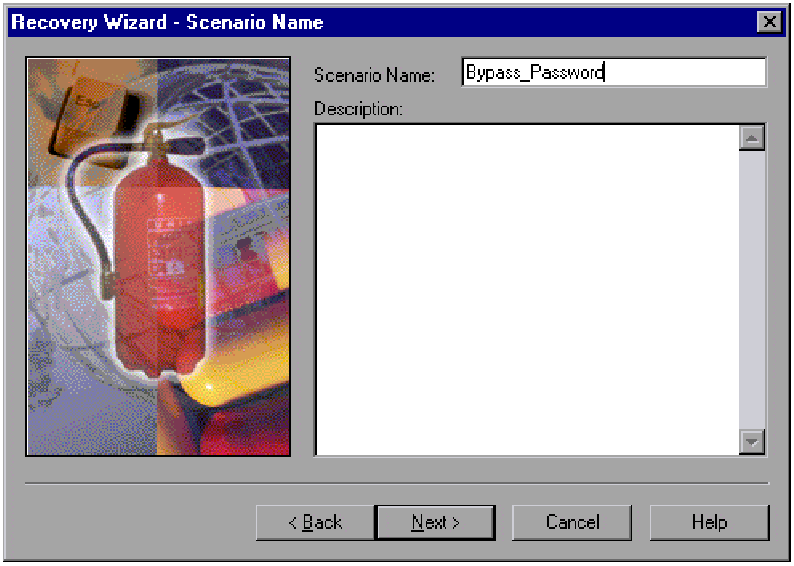
Notes:

By default, WinRunner identifies a crash event when a window opens containing the string: Application Error. You can modify the string that WinRunner uses to identify crash windows in the *excp_str.ini* file located in the <*WinRunner installation folder*>*dat* folder. For more information, see "Modifying the Crash Event Window Name" on page 78.

When you activate a crash recovery scenario, your tests may run more slowly. For more information, refer to the *WinRunner Readme*.

Click **Next**.

5 The Scenario Name screen opens.



Recovery Wizard - Scenario Name

Scenario Name: Bypass_Password

Description:

Enter a name containing only alphanumeric characters and underscores (no spaces or special characters) and a description for your recovery scenario.

Click **Next**.

6 If you chose an object, pop-up or TSL event in step 4, the Define Exception Event screen opens. The options for defining the event vary based on the type of event you selected.

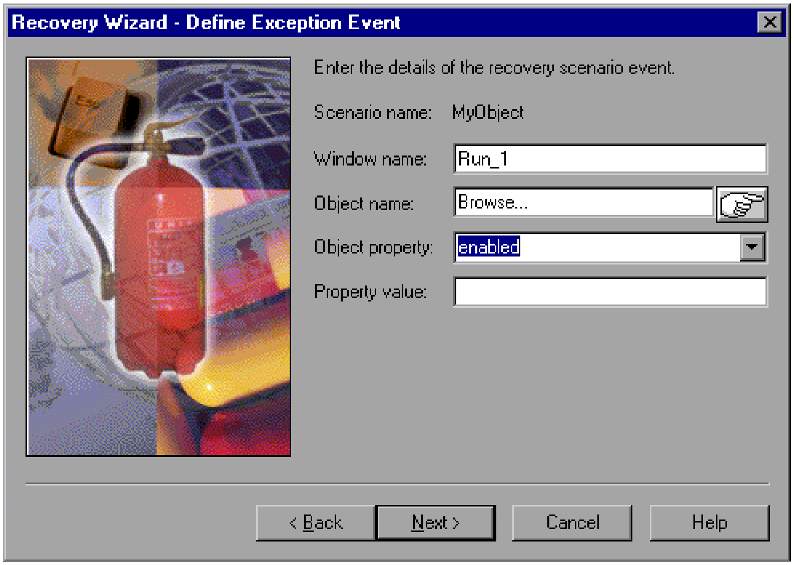
If you chose a crash event in step 4, there is no need to define the event. Proceed to step 7 on page 65.

For information on defining object events, see page 61.

For information on defining pop-up events, see page 63.

For information on defining TSL events, see page 64.

If you chose an object event in step 4, enter the following information:



Recovery Wizard - Define Exception Event

Enter the details of the recovery scenario event.

Scenario name: MyObject

Window name: Run_1

Object name: Browse...

Object property: enabled

Property value:

➤ **Window name:** Indicates the name of the window containing the object that causes the exception. Enter the logical name of the window, or use the pointing hand next to the **Object name** box to click on the object you want to define for the object exception and WinRunner will automatically fill in the Window name and Object name.

If you want to define a window as the exception object, click on the window's title bar, or enter the window's logical name and leave the **Object name** box empty.

- **Object name:** Indicates the name of the object that causes the exception. Enter the logical name of the object, or use the pointing hand next to the **Object name** box to specify the object you want to define for the object exception and WinRunner will automatically fill in the Window name and Object name.

Note: The object you define must be saved in the GUI Map. If the object is not already saved in the GUI Map and you use the pointing hand to identify the object, WinRunner automatically adds it to the active GUI Map. If you type the object name manually, you must also add the object to the GUI Map. For more information on the GUI Map, refer to Chapter 4, "Understanding Basic GUI Map Concepts" in the *Mercury WinRunner Basic Features User's Guide*.

- **Object property:** The object property whose value you want to check. Select the object property in which an exception may occur. For example, if want to detect when a button changes from enabled to disabled, select the enabled property.

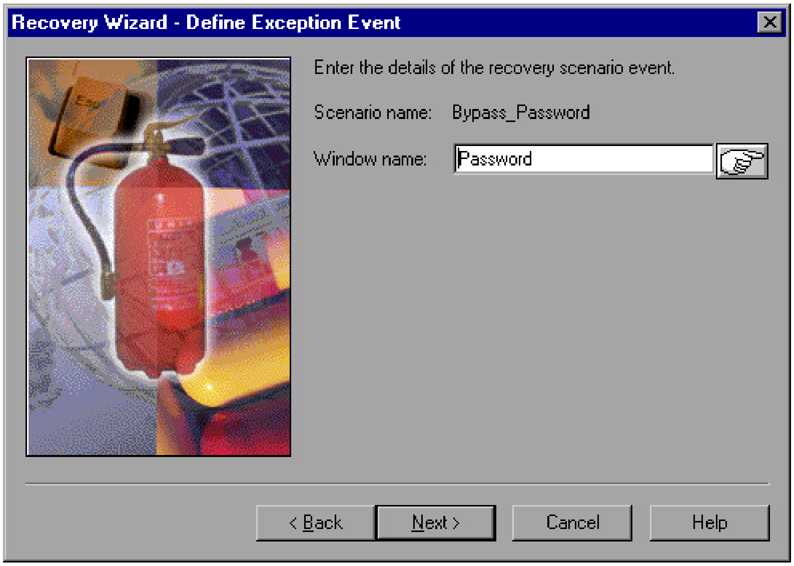
Note: You cannot specify a property that is part of the object's physical description.

- **Property value:** The value that indicates that an exception has occurred. For example, if you want WinRunner to activate the recovery scenario when the button changes from enabled to disabled, type 0 in the field.

Tip: Leave the property value empty to detect any change in the property value.

Click **Next** and proceed to step 7 on page 65.

If you chose a pop-up event in step 4, enter the following information:



Enter the details of the recovery scenario event.

Scenario name: Bypass_Password

Window name: Password

- **Window name:** Indicates the name of the pop-up window that causes the exception. Enter the logical name of the window, or use the pointing hand to specify the window you want to define as a pop-up exception.

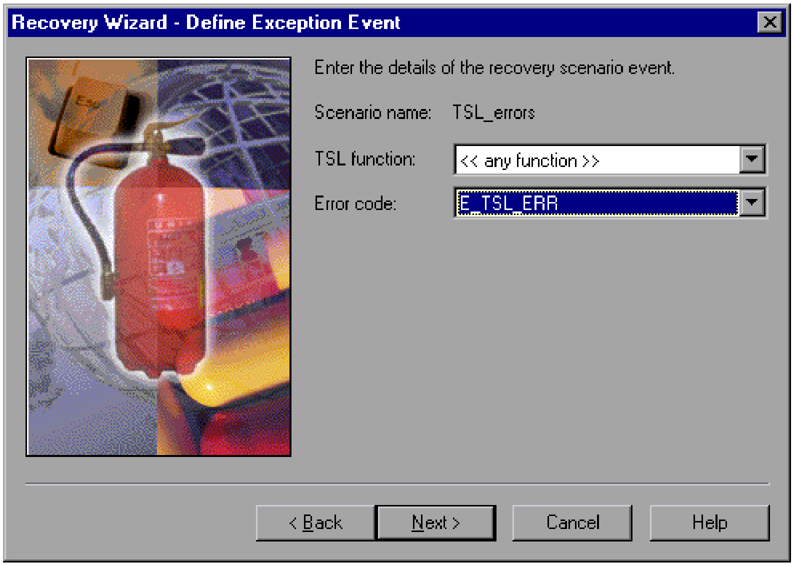
  If the window is not already saved in the GUI Map and you use the pointing hand to identify the window, WinRunner automatically adds it to the active GUI Map. If the window is not already saved in the GUI Map and you type the name manually, WinRunner identifies the pop-up exception when a pop-up window opens with a title bar matching the name you entered.

Note: If you want to employ a **Click button** recovery operation, then the pop-up window you define must be saved in the GUI Map. If you type the window name manually, you must also add the window to the GUI Map. For more information about recovery operations, see page 65.

Tip: If the pop-up window that causes the exception has a window name that is generated dynamically, use the pointing hand to add the window to the GUI Map and then modify the definition of the window in the GUI Map using regular expressions.

Click **Next** and proceed to step 7 on page 65.

If you chose a TSL event in step 4, enter the following information:



Recovery Wizard - Define Exception Event

Enter the details of the recovery scenario event.

Scenario name: TSL_errors

TSL function: << any function >>

Error code: E_TSL_ERR

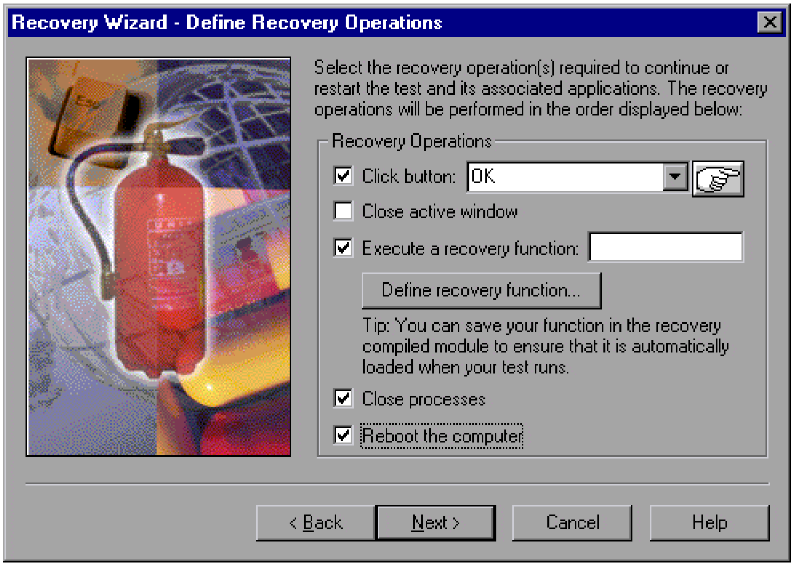
- **TSL function:** Select the TSL function for which you want to define the exception event. Select a TSL function from the list. WinRunner detects the exception only when the selected TSL function returns the code selected in the Error code box.

Tip: Select << **any function** >> to trigger the exception mechanism for any TSL function that returns the specified Error code.

- **Error code:** Select the TSL error code that triggers the exception mechanism. Select an error code from the list. WinRunner activates the recovery scenario when this return code is detected for the selected TSL function during a test run.

  Click **Next**.

7 The Define Recovery Operations screen opens and displays the recovery operations WinRunner can perform when the exception occurs.



Recovery Wizard - Define Recovery Operations

Select the recovery operation(s) required to continue or restart the test and its associated applications. The recovery operations will be performed in the order displayed below:

Recovery Operations

☑ Click button: OK

☐ Close active window

☑ Execute a recovery function:

Define recovery function...

Tip: You can save your function in the recovery compiled module to ensure that it is automatically loaded when your test runs.

☑ Close processes

☑ Reboot the computer

Note that WinRunner performs the recovery operations you select according to the order displayed in the dialog box. Select any of the following options:

- **Click button**: Specifies the logical name of the button to click when the exception event occurs. Select one of the default button names, type the logical name of a button, or use the pointing hand to specify the button to click.

Notes:

If you choose a default button from the list, the window on which WinRunner searches for the button depends on the type of exception event you selected. If you selected a pop-up exception event, WinRunner searches for the button on the pop-up window you defined. If you selected any other exception, then WinRunner searches for the button on the active (in focus) window.

When you use this option with a pop-up exception event, the pop-up window defined for the recovery scenario must be defined in the GUI map. If the pop-up window is not defined in a loaded GUI map file when you define the pop-up recovery scenario, the recovery scenario will automatically be set as inactive. If you later load a GUI map containing the pop-up window, you can then activate the recovery scenario.

➤ **Close active window:** Instructs WinRunner to close the active (in focus) window when the exception event occurs.

Note: WinRunner uses the (TSL) **win_close** mechanism to close the window. If the **win_close** function cannot close the window, the recovery scenario cannot close the window.

➤ **Execute a recovery function**: Instructs WinRunner to run the specified function when the exception event occurs. You can specify an existing function or click **Define recovery function** to define a new function. For more information on defining recovery functions, see “Defining Recovery Scenario Functions” on page 73.

Note: The compiled module containing the function must be loaded when the test runs. Save your function in the recovery compiled module to ensure that it is always automatically loaded when WinRunner opens. If you do not select a function saved in the recovery compiled module, ensure that the compiled module containing your function is loaded whenever a recovery scenario using the function is activated.

- **Close processes:** Instructs WinRunner to close the application processes that you specify in the Close Application Processes screen.
- **Reboot the computer:** Instructs WinRunner to reboot the computer before performing the post-recovery operations.

  If you select **Reboot the computer**, consider the following:

  - The reboot option is performed only after all other selected recovery actions have been performed.
  - In order to assure a smooth reboot process, it is recommended to use the **Execute a recovery function** option and add statements to your function that save any unsaved files before the reboot. You should also confirm that your computer is set to login automatically.

    Note: When a reboot occurs as part of a recovery scenario, tests open in WinRunner are automatically closed and you are not prompted to save changes.

  - If you choose the reboot option, you cannot set post-recovery operations.
  - Before WinRunner reboots the computer during a recovery scenario, you get a timed warning message that gives you a chance to cancel the reboot operation.

- If the reboot operation is performed, WinRunner starts running the test from the beginning of the test, or from the beginning of the call chain if the test that caused the exception was called by another test. For example, if test A calls test B, test B calls test C, and a recovery scenario including a reboot recovery operation is triggered when test C runs, WinRunner begins running test A from the beginning after the reboot is performed.
- If you choose to cancel the reboot operation, WinRunner attempts to continue the test from the point that the exception occurred.
- If you opened WinRunner using command line options before the reboot occurred, WinRunner applies the same command line options when it opens after the reboot operation, except for: -t, -exp, and -verify. Instead, WinRunner uses the test, expected values and results folder for the test it runs after the reboot.

Note: Recovery scenarios using a reboot recovery operation should not be activated when running tests from Quality Center, because WinRunner disconnects from Quality Center when a reboot occurs.

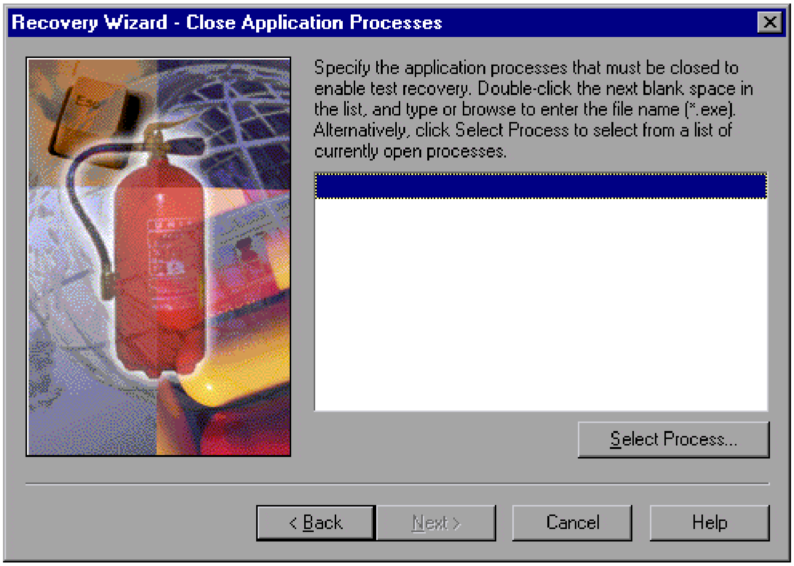
Click **Next**.

If you selected **Close processes**, proceed to step 8.

If you did not select **Close processes** or **Reboot the computer**, proceed to step 9.

If you selected **Reboot the computer**, but not **Close processes**, proceed to step 10.

8 The Close Application Processes screen opens.

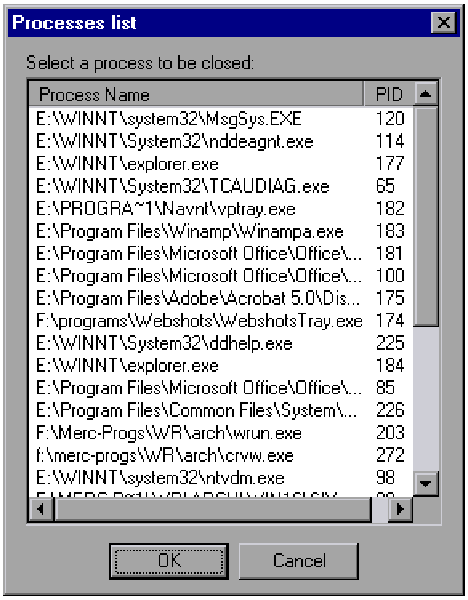


Recovery Wizard - Close Application Processes

Specify the application processes that must be closed to enable test recovery. Double-click the next blank space in the list, and type or browse to enter the file name (*.exe). Alternatively, click Select Process to select from a list of currently open processes.

Specify the application processes that you want WinRunner to close when the exception event occurs. When WinRunner runs the recovery scenario, it ignores listed application processes that are already closed (no error occurs).

To add an application to the list, double-click the next blank space on the list and type or browse to enter the application name, or click **Select Process** to open the Processes list. The Processes list contains a list of processes that are currently running.



Select a process to be closed:

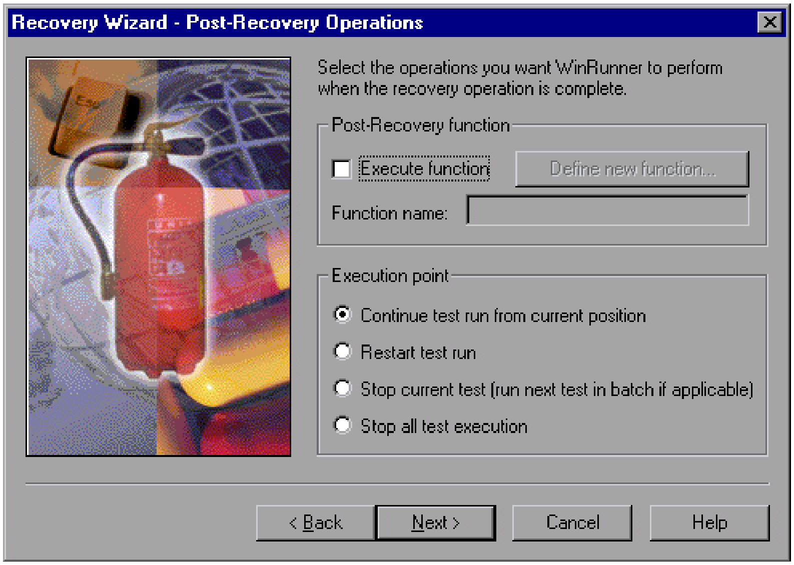
Process Name	PID
E:\WINNT\system32\MsgSys.EXE	120
E:\WINNT\System32\nddeagnt.exe	114
E:\WINNT\explorer.exe	177
E:\WINNT\System32\TCAUDIAG.exe	65
E:\PROGRA~1\Navnt\vptray.exe	182
E:\Program Files\Winamp\Winampa.exe	183
E:\Program Files\Microsoft Office\Office\...	181
E:\Program Files\Microsoft Office\Office\...	100
E:\Program Files\Adobe\Acrobat 5.0\Dis...	175
F:\programs\Webshots\WebshotsTray.exe	174
E:\WINNT\System32\ddhelp.exe	225
E:\WINNT\explorer.exe	184
E:\Program Files\Microsoft Office\Office\...	85
E:\Program Files\Common Files\System\...	226
F:\Merc-Progs\WR\arch\wrun.exe	203
f:\merc-progs\WR\arch\crvw.exe	272
E:\WINNT\system32\ntvdm.exe	98

To add a process from this list to the Close Application Processes list, select the process and click **OK**.

Note: The application names you specify must have *.exe* extensions.

Click **Next**. If you selected **Reboot the computer** in the previous step, proceed to step 10. Otherwise, proceed to step 9.

9 The Post-Recovery Operations screen opens.



Recovery Wizard - Post-Recovery Operations

Select the operations you want WinRunner to perform when the recovery operation is complete.

Post-Recovery function

□ Execute function

Function name:

Execution point

● Continue test run from current position

○ Restart test run

○ Stop current test (run next test in batch if applicable)

○ Stop all test execution

Choose from the following options:

- **Execute function:** Instructs WinRunner to run the specified function when the recovery operations are complete. You can specify an existing function or click **Define new function** to define a new function. For more information on defining post-recovery functions, see “Defining Recovery Scenario Functions” on page 73.

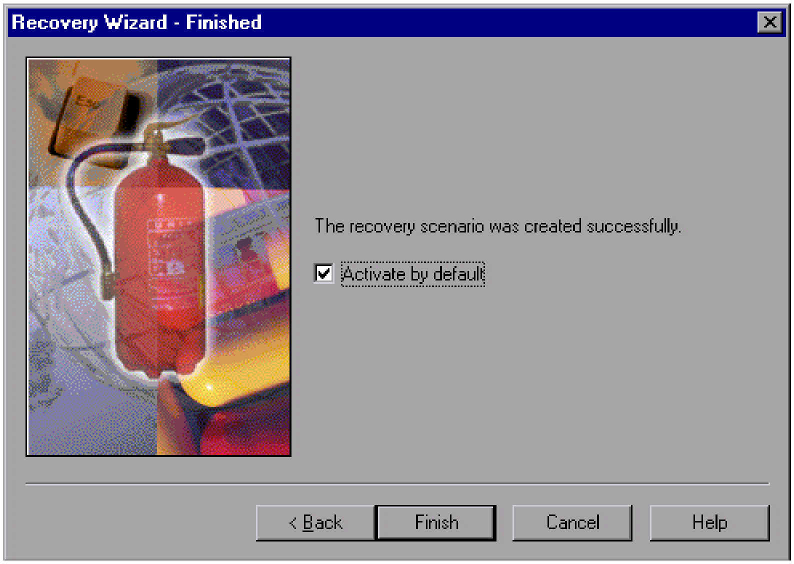
Tip: The compiled module containing the function must be loaded when the test runs. Save your function in the recovery compiled module to ensure that it is always automatically loaded when WinRunner opens. For more information on the recovery compiled module, see “Defining Recovery Scenario Functions” on page 73.

The post-recovery function can be useful for reopening applications that were closed during the recovery process and/or setting applications to the desired state.

- **Execution point:** Instructs WinRunner on how to proceed after the recovery operation(s) and the post-recovery function (if applicable) have been performed. Choose one of the following:
  - **Continue test run from current position:** WinRunner continues to run the current test from the location at which the exception occurred.
  - **Restart test run:** WinRunner runs the current test again from the beginning.
  - **Stop current test (run next test in batch if applicable)**: WinRunner stops the current test run. If the test where the exception event occurred was called from a batch test, WinRunner continues running the batch test from the next line in the test.
  - **Stop all test execution:** WinRunner stops the test (and batch) run.

Click **Next.**

10 The Finished screen opens.


Recovery Wizard - Finished

The recovery scenario was created successfully.

☑ Activate by default

Determine whether you want your recovery scenario to be activated by default when WinRunner opens:

➤ Select **Activate by default** to instruct WinRunner to automatically activate the recovery scenario by default when WinRunner opens, even if the scenario was set as inactive at the end of the previous WinRunner session.

➤ Clear **Activate by default** if you do not want WinRunner to automatically activate the recovery scenario by default when WinRunner opens. Note that if you clear this check box, your recovery scenario will not be activated unless you activate it manually by toggling the check box in the Recovery Manager dialog box. For more information, see "Activating and Deactivating Recovery Scenarios" on page 78.

Click **Finish**. The recovery scenario is added to the **Compound** tab of the Recovery Manager dialog box. If you selected Activate by default (and any required objects are found in the loaded GUI map file(s)), the recovery scenario is activated. Otherwise the recovery scenario remains inactive.

Defining Recovery Scenario Functions

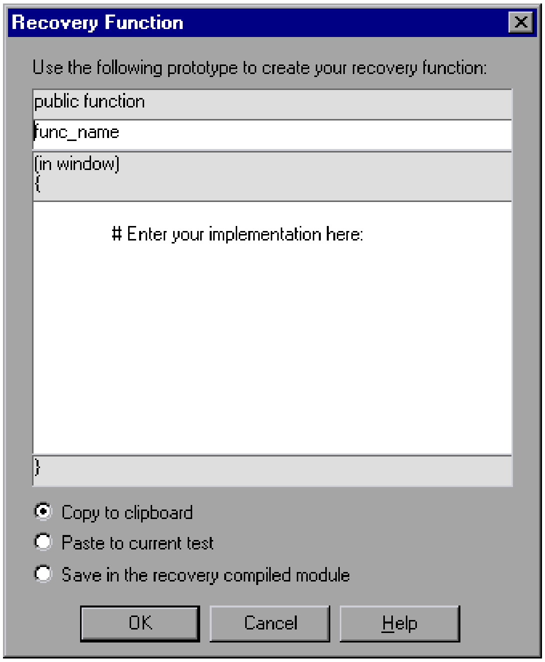
You can define recovery functions that instruct WinRunner to respond to an exception event in a way that meets your specific testing needs. You can also define post-recovery functions for compound recovery scenarios. These functions can be useful to re-open applications that may have closed when the exception occurred or during the recovery process, and to set applications to the desired state.

You use the Recovery Function or Post-Recovery Function dialog box that opens from the Recovery wizard to define new recovery and post-recovery functions. The dialog box displays the syntax and a function prototype for the selected exception type.

Once you have defined a recovery function, you can save it in the recovery compiled module, paste it into the current test, or copy it to the clipboard.

To define a recovery or post-recovery function:

1 Click **Define recovery function** from the Define Recovery Operations screen, or click **Define new function** from the Post-Recovery Operations screen. The Recovery (or Post-Recovery) Function screen opens.



Recovery Function

Use the following prototype to create your recovery function:

```
public function
func_name
(in window)
{

        # Enter your implementation here:

}
```

● Copy to clipboard

○ Paste to current test

○ Save in the recovery compiled module

2 The first three lines display the function type (always public function), the function name and the function arguments. Replace the text: func_name with the name of your new function.

3 In the implementation box, enter the function content.

4 Choose how you want to store the function:

- **Copy to clipboard:** copies the function to the clipboard.
- **Paste to current test:** pastes the function at the cursor position of the current test.
- **Save in the recovery compiled module:** saves the function in the recovery compiled module.

Notes:

If you have not defined a recovery compiled module in the **Run** > **Recovery** category of the General Options dialog box, the **Save in the recovery compiled module** option is disabled. For more information, see “Choosing the Recovery Compiled Module” on page 82.

If you save your function in the recovery compiled module, you must either restart WinRunner or run the compiled module manually in order to load the recovery compiled module with your changes before running tests that may require the new function.

If you do not select to save your function in the recovery compiled module, ensure that the compiled module containing your function is loaded whenever a recovery scenario using the function is activated.

5 Click **OK** to return to the Recovery wizard.

Managing Recovery Scenarios

Once you have created recovery scenarios, you can use the Recovery Manager to manage them. The Recovery Manager enables you to:

- View a summary of each recovery scenario
- Modify existing recovery scenarios using the Recovery wizard
- Activate or Deactivate existing recovery scenarios
- Delete Recovery scenarios

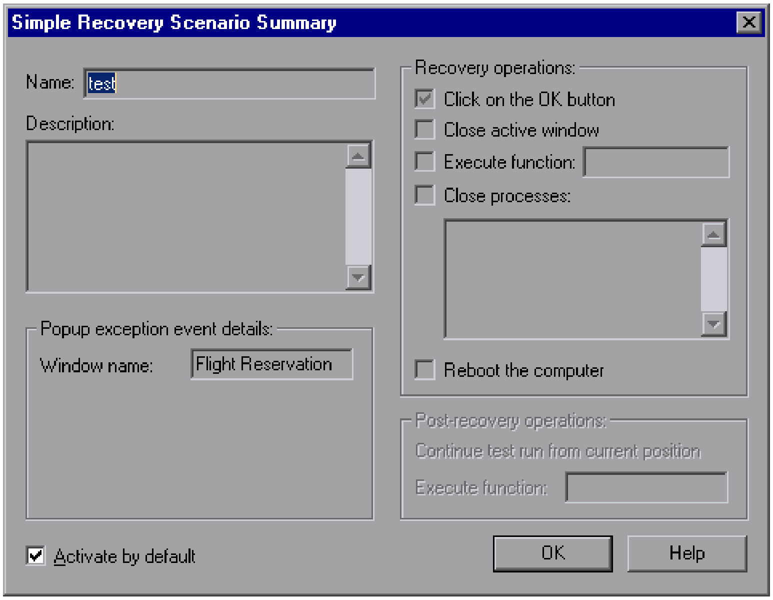
If you use crash recovery scenarios, you can also modify the string that WinRunner uses to identify crash windows.

Viewing Recovery Scenario Details

The Recovery Scenario Summary dialog box displays the details of the selected recovery scenario, and enables you to easily modify the Activate by default setting.

To open the Recovery Scenario Summary dialog box:

1 Select a recovery scenario in the **Simple** or **Compound** tab of the Recovery Manager dialog box, and click **Summary**, or double-click the recovery scenario name. The (Simple or Compound) Recovery Scenario Summary dialog box opens.



Simple Recovery Scenario Summary

Name: test

Description:

Popup exception event details:

Window name: Flight Reservation

Recovery operations:

☑ Click on the OK button

□ Close active window

□ Execute function:

□ Close processes:

□ Reboot the computer

Post-recovery operations:

Continue test run from current position

Execute function:

☑ Activate by default

2 Review the settings for the recovery scenario.

3 Select or clear the **Activate by default** check box if you want to modify the setting. For more information, see “Activating and Deactivating Recovery Scenarios” on page 78.

Modifying Recovery Scenarios

You can use the Modify option of the Recovery wizard to modify the details of an existing recovery scenario.

To modify a recovery scenario:

1 Select the recovery scenario you want to modify from the Recovery Manager dialog box and click **Modify**.

2 The Recovery wizard opens to the Scenario Name screen.

Note: You cannot modify the exception event type of an existing recovery scenario. If you want to define a different exception event type, create a new recovery scenario.

3 Navigate through the Recovery wizard and modify the details as needed. For information on the Recovery wizard options, see "Defining Simple Recovery Scenarios" on page 47 or "Defining Compound Recovery Scenarios" on page 58.

Deleting Recovery Scenarios

You can use the Delete option of the Recovery wizard to delete an existing recovery scenario. When you delete a recovery scenario from the Recovery Manager, the corresponding information is deleted from the recovery scenarios file.

For more information on the recovery scenarios file, refer to Chapter 23, "Setting Global Testing Options" in the *Mercury WinRunner Basic Features User's Guide*.

To delete a recovery scenario:

Select the recovery scenario you want to delete from the Recovery Manager dialog box and click **Delete**.

Note: Functions that you stored in the recovery compiled module when defining a recovery scenario are not deleted when you delete the recovery scenario. In order to control the size of the recovery compiled module, you should delete functions from the recovery compiled module if they are no longer being used by any recovery scenario.

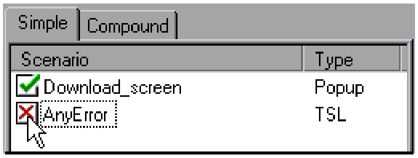
Activating and Deactivating Recovery Scenarios

WinRunner only identifies exception events and performs recovery operations for active recovery scenarios. You can activate or deactivate a recovery scenario in several ways:

- Select or clear the **Activate by default** check box when you create a recovery scenario.

☒ ☑

- Toggle (single-click) the activation check box next to the recovery scenario name in the Recovery Manager to *temporarily* activate or deactivate a recovery scenario. (The setting in the activate by default option resets the recovery scenario to its active or inactive state each time WinRunner opens.)



Scenario	Type
☑ Download_screen	Popup
☒ AnyError	TSL

- Select a recovery scenario in the Recovery Manager and click **Summary** or double-click the recovery scenario to open the Recovery Scenario Summary dialog box, and select or clear the **Activate by default** check box.
- Select a recovery scenario in the Recovery Manager, click **Modify** to open the Recovery wizard, navigate to the Finished screen and select or clear the **Activate by default** check box.
- Activate a recovery scenario during the test run using TSL commands. For more information on these functions, see “Working with Recovery Scenarios in Your Test Script” on page 84.

Modifying the Crash Event Window Name

WinRunner identifies a crash event when a window opens whose title bar contains the string indicating an application crash. You can modify the string that WinRunner uses to identify crash windows in the *excp_str.ini* file located in the <*WinRunner installation folder*>*dat* folder.

The *excp_str.ini* file is composed of sections for various Windows languages, plus a default section for unlisted languages. WinRunner uses the string corresponding to your Windows language to identify a crash event.

To modify the crash event window name, modify the window name listed in the section corresponding to the Windows language you are using. The language sections in the *excp_str.ini* file are identified by the three letter LOCALE_SABBREVLANGNAME code.

If your Windows language is not listed, enter the crash event string you want to use in the [DEF] section. Alternatively, add a new section to the file using the three letter LOCALE_SABBREVLANGNAME for your Windows Language as the section divider, and enter the crash event string below it in quotes ("*string*").

The table below lists each of the codes contained in the *excp_str.ini* by default and the corresponding Windows language. For the complete list of language codes, refer to MSDN documentation.

Language Code	Windows Language
ENU	English (U.S.)
JPN	Japanese
KOR	Korean
CHS	Chinese (PRC)
CHT	Chinese (Taiwan)
DEU	German (Germany)
SVE	Swedish (Sweden)
FRA	French (France)

Working with Recovery Scenarios Files

When you create, modify or delete recovery scenarios, the information is saved in the active recovery scenarios file. Each time WinRunner opens, it reads the information in the active file and includes the recovery scenarios that are defined in the file in the Recovery Manager. You can create multiple recovery scenarios files and then select different recovery scenarios files for different WinRunner sessions as needed.

Note: The recovery files are used only to store the recovery information so that you can alternate between various recovery scenario configurations. You use the Recovery Manager and recovery wizard to create, modify, or delete recovery scenarios.

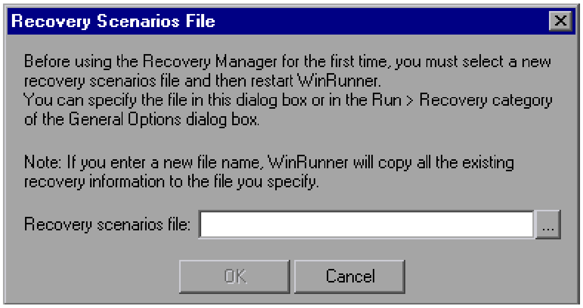
Using the Recovery Manager for the First Time

In WinRunner, version 7.01 and earlier, all “exception handling” details were saved in the wrun.ini file. Therefore, the *wrun.ini* file is the default recovery scenarios file.

When you open the Recovery Manager for the first time, any exceptions defined in the *wrun.ini* file are displayed in the **Simple** tab of the Recovery Manager and they work as they did in previous versions of WinRunner.

In order to create, modify, or delete recovery scenarios using the Recovery Manager, however, you must define a new recovery scenarios file.

You can enter a file name in the dialog that opens the first time you try to create, modify, or delete a recovery scenario.


Recovery Scenarios File

Before using the Recovery Manager for the first time, you must select a new recovery scenarios file and then restart WinRunner.
You can specify the file in this dialog box or in the Run > Recovery category of the General Options dialog box.

Note: If you enter a new file name, WinRunner will copy all the existing recovery information to the file you specify.

Recovery scenarios file:

Alternatively, you can define the new recovery scenarios file in the **Run** > **Recovery** category of the General Options dialog box before using the Recovery Manager for the first time.

If you enter a new file name, WinRunner creates the file and any exceptions information that was previously contained in the *wrun.ini* file is copied to the new file so that you can continue to work with your existing exception handling definitions using the Recovery Manager. For more information on recovery scenarios files and how to choose them, see “Choosing the Active Recovery Scenarios File” below.

Choosing the Active Recovery Scenarios File

You select the active recovery scenarios file in **Run** > **Recovery** category in the General Options dialog box. You can select an existing file or enter a new file name.

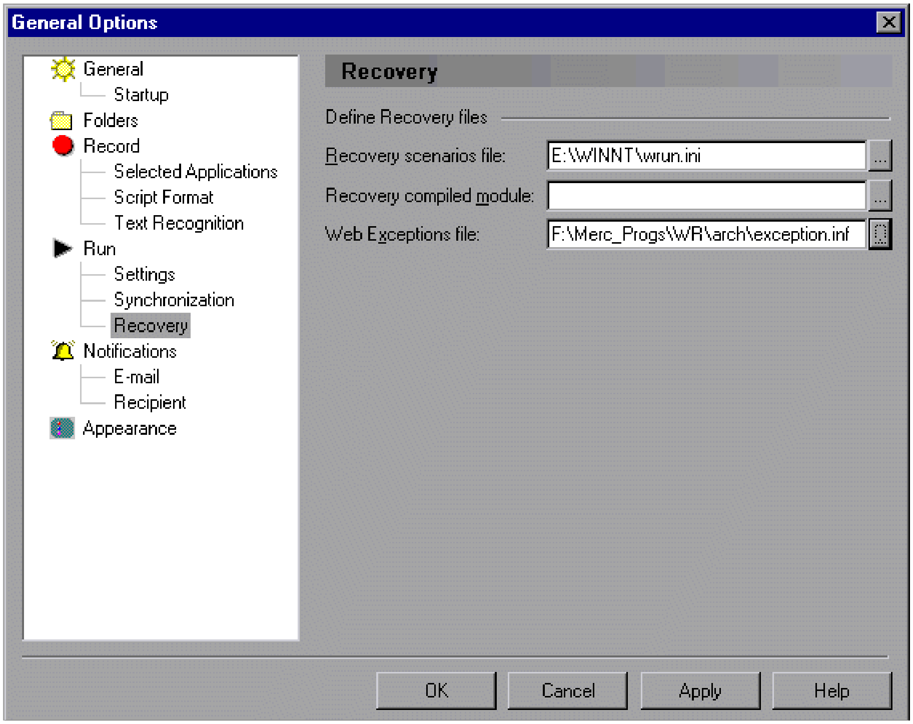
When you enter a new file name and confirm that you want WinRunner to create the new file, WinRunner copies all recovery scenario information from the current recovery scenarios file to the new file.

When you enter the name of an existing recovery scenarios file, WinRunner sets the selected file as the active recovery scenarios file, but does not copy any information from the previous recovery scenarios file.

To select an active recovery scenarios file:

1 Choose **Tools** > **General Options**.

2 Click the **Run > Recovery** category. The Recovery options pane is displayed.



General Options

Recovery

Define Recovery files

Recovery scenarios file: E:\WINNT\wrun.ini

Recovery compiled module:

Web Exceptions file: F:\Merc_Progs\WR\arch\exception.inf

3 In the **Recovery scenarios file** box, type the path of the file you want to use (or create), or click browse to select an existing recovery scenarios file.

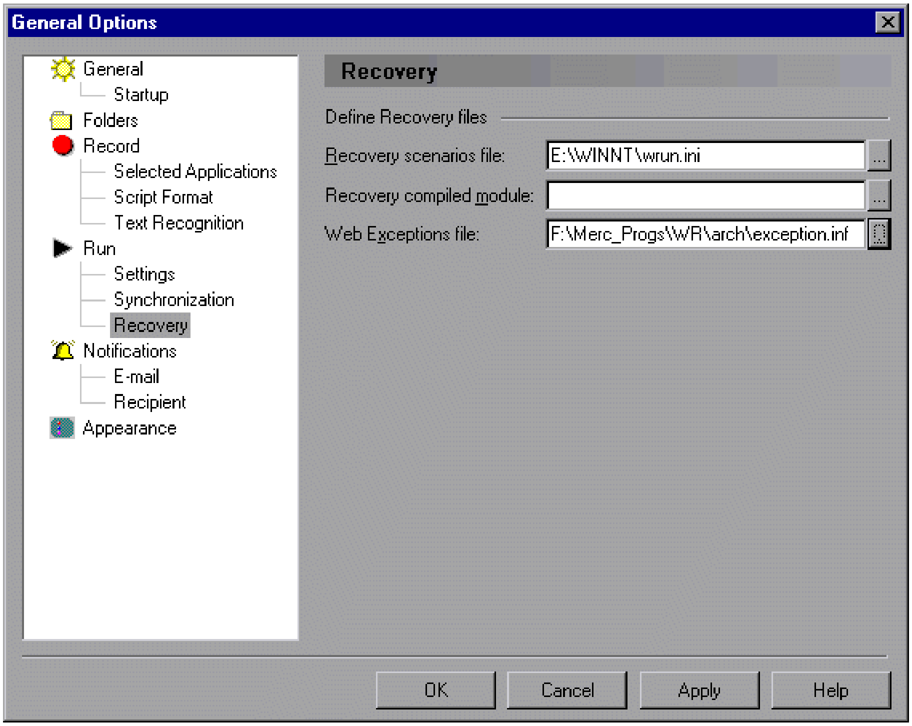
Choosing the Recovery Compiled Module

The recovery compiled module is a special compiled module that is always loaded when WinRunner opens so that the functions it contains can be accessed whenever WinRunner performs a recovery scenario.

You can instruct WinRunner to save the functions you define in the Define Recovery Function or Define Post-Recovery Function dialog boxes directly to the recovery compiled module while creating or editing a recovery scenario. You can also open the recovery compiled module and add functions to the compiled module manually.

To select an active recovery compiled module

1. Choose **Tools** > **General Options**.
2. Click the **Run** > **Recovery** category. The Recovery options pane is displayed.



General Options

Recovery

Define Recovery files

Recovery scenarios file: E:\WINNT\wrun.ini

Recovery compiled module:

Web Exceptions file: F:\Merc_Progs\WR\arch\exception.inf

3. In the **Recovery compiled module** box, type the path of the compiled module you want to use (or create), or click browse to select an existing compiled module. If you enter a new file name, WinRunner creates a new compiled module.

   For more information on compiled modules, see Chapter 11, “Employing User-Defined Functions in Tests.”

   For more information on the selecting the recovery compiled module file, refer to Chapter 23, “Setting Global Testing Options” in the *Mercury WinRunner Basic Features User’s Guide*.

Working with Recovery Scenarios in Your Test Script

You can use TSL statements to activate or deactivate a specific recovery scenario, or to deactivate all active recovery scenarios during a test run. You can also define simple recovery scenarios using TSL.

Activating and Deactivating Recovery Scenarios During the Test Run

The Recovery Manager enables you to activate or deactivate recovery scenarios while designing your test, but you may need to turn a recovery scenario on or off during a test run.

Suppose you define a recovery scenario that runs a recovery function. If the exception event triggers the recovery scenario, and then the exception event occurs again while the recovery function for that event is running, the recovery scenario may get stuck in an infinite loop. Thus it is recommended to deactivate the recovery scenario at the beginning of that recovery scenario's recovery function, and to reactivate it at the end of the function.

To activate and deactivate a specific recovery scenario use the **exception_on** and **exception_off** functions.

For example: The following recovery function turns off the handling of its recovery scenario before executing the main recovery script (which reopens the application being tested). Then it turns the recovery scenario on again.

```
public function label_handler(in win, in obj, in attr, in val)
{
#ignore this recovery scenario while performing the recovery function:
exception_off("label_except");
report_msg("Label has changed");
menu_select_item ("File;Exit");
system ("flights&");
invoke_application ("flights", "", "C:\\FRS", "SW_SHOWMAXIMIZED");
#if the value of "attr" no longer equals "val":
exception_on("label_except");
texit;
}
```

You can also deactivate all recovery scenarios during a test run. For example, you may want to prevent WinRunner from performing recovery scenarios if a certain conditional statement is true.

To deactivate all active recovery scenarios, use the **exception_off_all** function.

For more information on these functions, refer to the *TSL Reference*.

Defining Simple Recovery Scenarios Using TSL

You can use the **define_object_exception**, **define_popup_exception**, and **define_TSL_exception** functions to define new simple recovery scenarios from your test script that are active only for the current WinRunner session. This can be useful if you want to use a returned value as input for your recovery scenario.

When you define a simple recovery scenario using one of the above functions, the simple recovery scenario is displayed in the Recovery Manager during the WinRunner session and you can modify the recovery scenario using the recovery wizard, but these recovery scenarios are not saved in the recovery scenarios file and are not available from the Recovery Manager when WinRunner restarts.

To create compound recovery scenarios, which enable you to define crash events and/or multiple recovery operations, use the Recovery Manager. For more information, see “Defining Compound Recovery Scenarios” on page 58.

For more information on defining simple recovery scenarios using TSL, refer to the *TSL Reference*.

5

Handling Web Exceptions

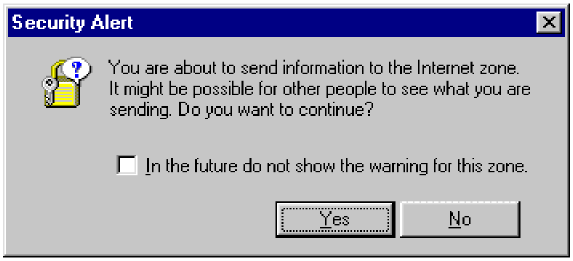
You can instruct WinRunner to handle unexpected events and errors that occur in your testing environment while testing your Web site.

This chapter describes:

- About Handling Web Exceptions
- Defining Web Exceptions
- Modifying an Exception
- Activating and Deactivating Web Exceptions

About Handling Web Exceptions

When the WebTest add-in is loaded, you can instruct WinRunner to handle unexpected events and errors that occur in your Web site during a test run. For example, if a Security Alert dialog box appears during a test run, you can instruct WinRunner to recover the test run by clicking the **Yes** button.



Security Alert

You are about to send information to the Internet zone. It might be possible for other people to see what you are sending. Do you want to continue?

□ In the future do not show the warning for this zone.

WinRunner contains a list of exceptions that it supports in the Web Exception Editor. You can modify the list and configure additional exceptions that you would like WinRunner to support.

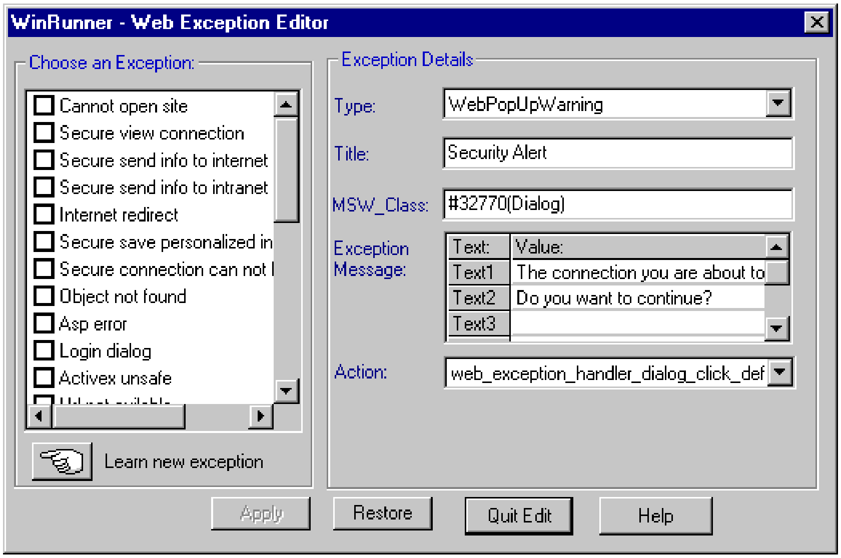
For information on loading WinRunner with the WebTest add-in, refer to Chapter 2, “WinRunner at a Glance” in the *Mercury WinRunner Basic Features User’s Guide*.

Defining Web Exceptions

You can add a new exception to the list of exceptions in the Web Exception Editor.

To define a Web exception:

1 Choose **Tools** > **Web Exception Handling**. The Web Exception Editor opens.



- ☐ Cannot open site
- ☐ Secure view connection
- ☐ Secure send info to internet
- ☐ Secure send info to intranet
- ☐ Internet redirect
- ☐ Secure save personalized in
- ☐ Secure connection can not l
- ☐ Object not found
- ☐ Asp error
- ☐ Login dialog
- ☐ Activex unsafe

Type: WebPopUpWarning

Title: Security Alert

MSW_Class: #32770(Dialog)

Exception Message:

Text:	Value:
Text1	The connection you are about to
Text2	Do you want to continue?
Text3	

Action: web_exception_handler_dialog_click_def

2 Click the pointing hand and click the dialog box. A new exception is added to the list.

3 If you want to categorize the exception, select a category in the **Type** list.

The Editor displays the title, MSW_Class, and message of the exception.

4 In the **Action** list, choose the handler function action that is responsible for recovering test execution.

   ➤ **Web_exception_handler_dialog_click_default** activates the default button.

   ➤ **Web_exception_handler_fail_retry** activates the default button and reloads the Web page.

   ➤ **Web_exception_enter_username_password** uses the given user name and password.

   ➤ **Web_exception_handler_dialog_click_yes** activates the **Yes** button.

   ➤ **Web_exception_handler_dialog_click_no** activates the **No** button.

5 Click **Apply.** The Save Configuration message box opens.

6 Click **OK** to save the changes to the configuration file.

7 Click **Quit Edit** to exit the Web Exception Editor.

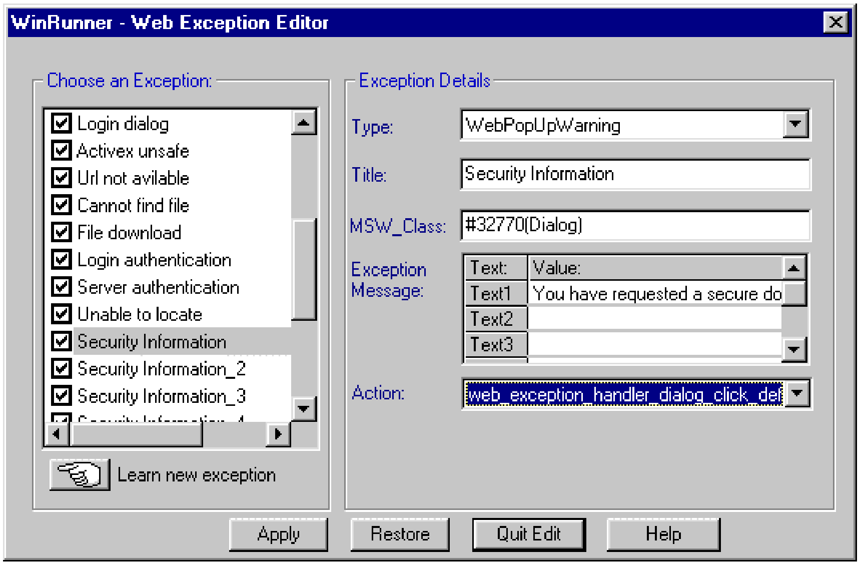
Modifying an Exception

You can modify the details of an exception listed in the Web Exception Editor.

To modify the details of an exception:

1 Choose **Tools** > **Web Exception Handling**. The Web Exception Editor opens.

2 In the **Choose an Exception** list, click an exception.



WinRunner - Web Exception Editor

Choose an Exception:

- ☑ Login dialog
- ☑ Activex unsafe
- ☑ Url not avilable
- ☑ Cannot find file
- ☑ File download
- ☑ Login authentication
- ☑ Server authentication
- ☑ Unable to locate
- ☑ Security Information
- ☑ Security Information_2
- ☑ Security Information_3

Exception Details

Type: WebPopUpWarning

Title: Security Information

MSW_Class: #32770(Dialog)

Exception Message:

Text:	Value:
Text1	You have requested a secure do
Text2	
Text3	

Action: web_exception_handler_dialog_click_de

The exception is highlighted. The current description of the exception appears in the Exception Details area.

3 To modify the title of the dialog box, type a new title in the **Title** box.

4 To modify the text that appears in the exception dialog box, click a text line and edit the text.

5 To change the action that is responsible for recovering test execution, choose an action from the **Action** list.

   ➤ **Web_exception_handler_dialog_click_default** activates the default button.

   ➤ **Web_exception_handler_fail_retry** activates the default button and reloads the Web page.

   ➤ **Web_exception_enter_username_password** uses the given user name and password.

   ➤ **Web_exception_handler_dialog_click_yes** activates the **Yes** button.

   ➤ **Web_exception_handler_dialog_click_no** activates the **No** button.

6 Click **Apply.** The Save Configuration message box opens.

7 Click **OK** to save the changes to the configuration file.

8 Click **Quit Edit** to exit the Web Exception Editor.

Activating and Deactivating Web Exceptions

The Web Exception Editor includes a list of all the available exceptions. You can choose to activate or deactivate any exception in the list.

To change the status of an exception:

1 Choose **Tools** > **Web Exception Handling**. The Web Exception Editor opens.

2 In the **Choose an Exception** list, click an exception. The exception is highlighted. The current description of the exception appears in the Exception Details area.

3 To activate an exception, select its check box. To deactivate the exception, clear its check box.

4 Click **Apply**. The Save Configuration message box opens.

5 Click **OK** to save the changes to the configuration file.

6 Click **Quit Edit** to exit the Web Exception Editor.

6

Using Regular Expressions

You can use regular expressions to increase the flexibility and adaptability of your tests. This chapter describes:

- About Regular Expressions
- Understanding When to Use Regular Expressions
- Understanding Regular Expression Syntax

About Regular Expressions

Regular expressions enable WinRunner to identify objects with varying names or titles. You can use regular expressions in TSL statements or in object descriptions in the GUI map. For example, you can define a regular expression in the physical description of a push button so that WinRunner can locate the push button if its label changes.

A regular expression is a string that specifies a complex search phrase. In most cases the string is preceded by an exclamation point (!). (In descriptions or arguments of functions where a string is expected, such as the **match** function, the exclamation point is not required.) By using special characters such as a period (.), asterisk (*), caret (^), and brackets ([ ]), you define the conditions of the search. For example, the string “!windo.*” matches both “window” and “windows”. See “Understanding Regular Expression Syntax” on page 97 for more information.

Understanding When to Use Regular Expressions

Use a regular expression when the name of a GUI object can vary each time you run a test. For example, you can use a regular expression for:

- the physical description of an object in the GUI map
- a GUI checkpoint, when evaluating the contents of an edit object or static text object with a varying name
- a text checkpoint, to locate a varying text string using **win_find_text** or **obj_find_text**

Using a Regular Expression in an Object's Physical Description in the GUI Map

You can use a regular expression in the physical description of an object in the GUI map, so that WinRunner can ignore variations in the object's label. For example, the physical description:

```
{
class: push_button
label: "!St.*"
}
```

enables WinRunner to identify a push button if its label toggles from "Start" to "Stop".

Using a Regular Expression in a GUI Checkpoint

You can use a regular expression in a GUI checkpoint, when evaluating the contents of an edit object or a static text object with a varying name. You define the regular expression by creating a GUI checkpoint on the object in which you specify the checks. The example below illustrates how to use a regular expression if you choose **Insert** > **GUI Checkpoint** > **For Object/Window** and double-click a static text object. Note that you can also use a regular expression with the **Insert** > **GUI Checkpoint** > **For Multiple Objects** command. For additional information about GUI checkpoints, refer to Chapter 9, "Checking GUI Objects" in the *Mercury WinRunner Basic Features User's Guide*.

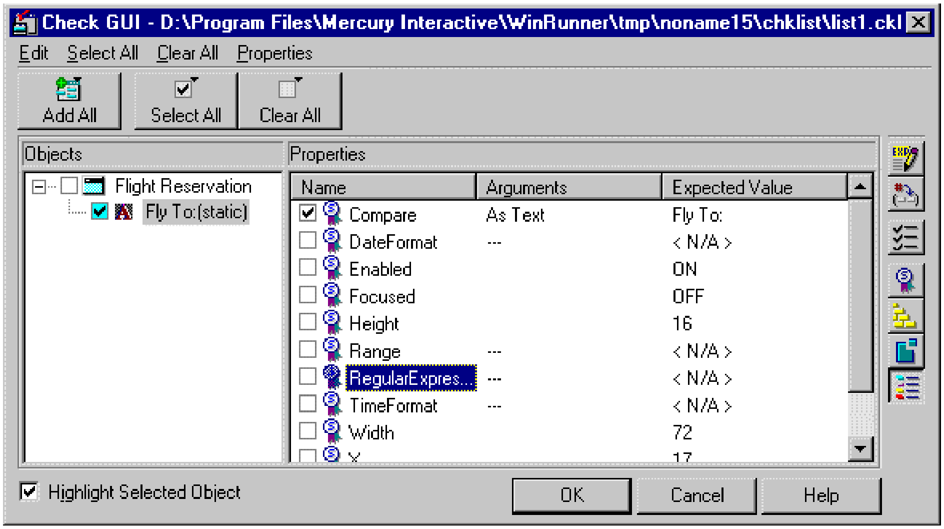
To define a regular expression in a GUI checkpoint:

1. Create a GUI checkpoint for an object in which you specify the checks. In this example, choose **Insert** > **GUI Checkpoint** > **For Object/Window**.

   The WinRunner window is minimized, the mouse pointer becomes a pointing hand, and a help window opens on the screen.

2. Double-click a static text object.

3. The Check GUI dialog box opens:


Check GUI - D:\Program Files\Mercury Interactive\WinRunner\tmp\noname15\chklist\list1.ckl
Edit
Select All
Clear All
Properties
Add All
Select All
Clear All
Objects
Properties
OK
Cancel
Help

Objects:

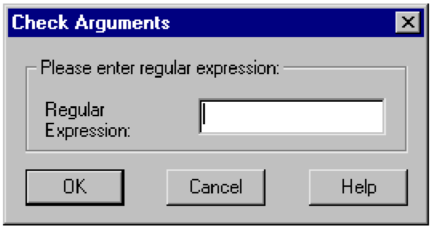
- ☐ Flight Reservation
  - ☑ Fly To:(static)

Name	Arguments	Expected Value
☑ Compare	As Text	Fly To:
☐ DateFormat	---	< N/A >
☐ Enabled		ON
☐ Focused		OFF
☐ Height		16
☐ Range	---	< N/A >
☐ RegularExpres...	---	< N/A >
☐ TimeFormat	---	< N/A >
☐ Width		72
☐ X		17

☑ Highlight Selected Object

4. In the **Properties** pane, highlight the “Regular Expression” property check and then click the **Specify Arguments** button.

   The Check Arguments dialog box opens:



Please enter regular expression:

Regular Expression:

5 Enter the regular expression in the **Regular Expression** box, and then click **OK**.

Note: When a regular expression is used to perform a check on a static text or edit object, it should *not* be preceded by an exclamation point.

---

6 If desired, specify any additional checks to perform, and then click **OK** to close the Check GUI dialog box.

An **obj_check_gui** statement is inserted into your test script.

For additional information on specifying arguments, refer to Chapter 9, “Checking GUI Objects” in the *Mercury WinRunner Basic Features User’s Guide*.

Using a Regular Expression in a Text Checkpoint

You can use a regular expression in a text checkpoint, to locate a varying text string using **win_find_text** or **obj_find_text**. For example, the statement:

```
obj_find_text ("Edit", "win.*", coord_array, 640, 480, 366, 284);
```

enables WinRunner to find any text in the object named “Edit” that begins with “win”.

Since windows often have varying labels, WinRunner defines a regular expression in the physical description of a window. For more information, refer to Chapter 7, “Editing the GUI Map” in the *Mercury WinRunner Basic Features User’s Guide*.

Understanding Regular Expression Syntax

Regular expressions must begin with an exclamation point (!), except when defined in a Check GUI dialog box, a text checkpoint, or a **match**, **obj_find_text**, or **win_find_text** statement. All characters in a regular expression are searched for literally, except for a period (.), asterisk (*), caret (^), and brackets ([ ]), as described below. When one of these special characters is preceded by a backslash (\), WinRunner searches for the literal character. For example, if you are using a **win_find_text** statement to search for a phrase beginning with “Sign up now!”, then you should use the following regular expression: “Sign up now\!.”

The options described in the remainder of this chapter can be used to create regular expressions.

Matching Any Single Character

A period (.) instructs WinRunner 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 brackets ([ ]). For example, to search for a date that 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:

```
{
class: static_text,
label: "!Quantity[0-9]"
}
```

In the above example, WinRunner can identify the static_text object with the label “Quantity” when the quantity number varies.

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 WinRunner 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 “]“ and g through m.

Note: Two “\” characters together (“\\”) are interpreted as a single “\” character. For example, in the physical description in a GUI map, “!D:\\.*” does not mean all labels that start with “D:\”. Rather, it refers to all labels that start with “D:.”. To specify all labels that start with “D:\”, use the following regular expression: “!D:\\\\.*”.

Matching Specific Characters

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

```
Q*
```

causes WinRunner to match Q, QQ, QQQ, etc.

A period "." followed by an asterisk "*" instructs WinRunner to locate the specific characters followed by any combination of characters.

For example, in the following physical description, the regular expression enables WinRunner to locate any push button that starts with "O" (for example, On or Off):

```
{
class: push_button
label: "!O.*"
}
```

You can also use a combination of brackets and an asterisk to limit the label to a combination of non-numeric characters. For example:

```
{
class: push_button
label: "!O[a-zA-Z]*"
}
```

Part III

Programming with TSL

7

Enhancing Your Test Scripts with Programming

WinRunner test scripts are composed of statements coded in Mercury Interactive's Test Script Language (TSL). This chapter provides a brief introduction to TSL and shows you how to enhance your test scripts using a few simple programming techniques.

This chapter describes:

About Enhancing Your Test Scripts with Programming

When you record a test, a test script is generated in Mercury Interactive's Test Script Language (TSL). Each line WinRunner generates in the test script is a *statement*. A statement is any expression that is followed by a semicolon. Each TSL statement in the test script represents keyboard and/or mouse input to the application being tested. A single statement may be longer than one line in the test script.

For example:

```
if (button_check_state("Underline", OFF) == E_OK)
    report_msg("Underline check box is unavailable.");
```

TSL is a C-like programming language designed for creating test scripts. It combines functions developed specifically for testing with general purpose programming language features such as variables, control-flow statements, arrays, and user-defined functions. You enhance a recorded test script simply by typing programming elements into the test window, If you program a test script by typing directly into the test window, remember to include a semicolon at the end of each statement.

TSL is easy to use because you do not have to compile. You simply record or type in the test script and immediately execute the test.

TSL includes four types of functions:

- *Context Sensitive* functions perform specific tasks on GUI objects, such as clicking a button or selecting an item from a list. Function names, such as **button_press** and **list_select_item**, reflect the function's purpose.
- *Analog* functions depict mouse clicks, keyboard input, and the exact coordinates traveled by the mouse.
- *Standard* functions perform general purpose programming tasks, such as sending messages to a report or performing calculations.
- *Customization* functions allow you to adapt WinRunner to your testing environment.

WinRunner includes a visual programming tool which helps you to quickly and easily add TSL functions to your tests. For more information, see Chapter 8, "Generating Functions."

This chapter introduces some basic programming concepts and shows you how to use a few simple programming techniques in order to create more powerful tests. For more information, refer to the following documentation:

- The *TSL Reference* includes general information about the TSL language, individual functions, examples of usage, function availability, and guidelines for working with TSL. You can open this online reference by choosing **Help** > **TSL Reference**. You can also open this reference directly to the help topic for a specific function by pressing the F1 key when your cursor is on a TSL statement in your test script, or by clicking the context-sensitive Help button and then clicking a TSL statement in your test script.
- The *TSL Reference Guide* includes general information about the TSL language, individual functions, function availability, and guidelines for working with TSL. This printed book is included in your WinRunner documentation set. You can also access a PDF version of this book, which is easy to print, by choosing **Help** > **Books Online** and then clicking **Test Script Language** from the WinRunner Books Online home page.

Using Descriptive Programming

When you add an object to the GUI Map, WinRunner assigns it a logical name. Once an object exists in the GUI Map, you can add statements to your test that perform functions on that object. To add these statements, you usually enter the logical name of the object as the object description.

For example, in the statements below, Flight Reservation is the logical name of a window, and File;Open Order is the logical name of the menu.

```
set_window ("Flight Reservation", 5);
menu_select_item ("File;Open Order...");
```

Because each object in the GUI Map has a unique logical name, this is all you need to describe the object. During the test run, WinRunner finds the object in the GUI Map based on its logical name and uses the other property values stored for that object to identify the object in your application.

Note: You can modify the logical name of any object in the GUI Map to make it easier for you to identify in your test. For more information, refer to Chapter 7, “Editing the GUI Map” in the *Mercury WinRunner Basic Features User’s Guide*.

You can also add statements to perform functions on objects without referring to the GUI Map. To do this, you need to enter more information in the description of the object in order to uniquely describe the object so that WinRunner can identify the object during the test run. This is known as *descriptive programming*.

For example, suppose you recorded a purchase order in a flight reservation application. Then, after you created your test, an additional radio button group was added to the purchase order. Rather than recording a new step in your existing test in order to add to the object to the GUI Map, you can add a statement to the script that describes the radio button you want to select, and sets the radio button state to ON.

You describe the object by defining the object class, the MSW_class, and as many additional *property:value* pairs as necessary to uniquely identify the object.

The general syntax is:

*function_name***("{ class:** *class_value* , **MSW_class:** *MSW_value , property3: value , ... , propertyX: value* **}" ,** *other_function_parameters***) ;**

function_name: The function you want to perform on the object.

property:value: The object property and its value. Each property:value pair should be separated by commas.

other_function_parameters: You enter other required or optional function parameters in the statement just as you would when using the logical name for the object parameter.

The entire object description should surrounded by curly brackets and quotes: “{description}”.

For example, the statement below performs a **button_set** function on a radio button to select a business class airline seat. When the test runs, WinRunner finds the radio button object with matching property values and selects it."

```
set_window ("Flight Reservation", 3);
button_set ("{class: radio_button, MSW_class: Button, label: Business}", ON);
```

If you are not sure which properties and values you can use to identify an object, use the GUI Spy to view the current properties and values of the object. For more information, refer to Chapter 4, "Understanding Basic GUI Map Concepts" in the *Mercury WinRunner Basic Features User's Guide*.

Adding Comments and White Space

When programming, you can add comments and white space to your test scripts to make them easier to read and understand.

Using Comments

A comment is a line or part of a line in a test script that is preceded by a pound sign (#). When you run a test, the TSL interpreter does not process comments. Use comments to explain sections of a test script in order to improve readability and to make tests easier to update.

For example:

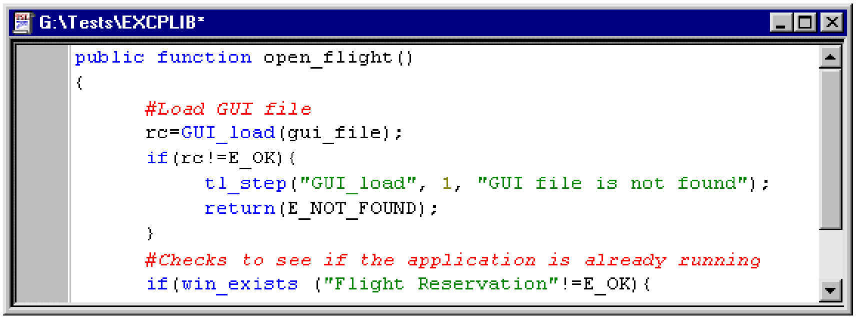
```
# Open the Open Order window in Flight Reservation application
set_window ("Flight Reservation", 10);
menu_select_item ("File;Open Order...");

# Select the reservation for James Brown
set_window ("Open Order_1");
button_set ("Customer Name", ON);
edit_set ("Value", "James Brown"); # Type James Brown
button_press ("OK");
```

You can use the **Insert comments and indent statements** option in the **Record** > **Script Format** category of the General Options dialog box to have WinRunner automatically divide your test script into sections while you record based on window focus changes. When you choose this option, WinRunner automatically inserts a comment at the beginning of each section with the name of the window and indents the statements under each comment. For more information on the Insert comments and indent statements option, refer to Chapter 23, “Setting Global Testing Options” in the *Mercury WinRunner Basic Features User’s Guide*.

Inserting White Space

White space refers to spaces, tabs, and blank lines in your test script. The TSL interpreter is not sensitive to white space unless it is part of a literal string. Use white space to make the logic of a test script clear.



```
public function open_flight()
{
	#Load GUI file
	rc=GUI_load(gui_file);
	if(rc!=E_OK){
		tl_step("GUI_load", 1, "GUI file is not found");
		return(E_NOT_FOUND);
	}
	#Checks to see if the application is already running
	if(win_exists ("Flight Reservation"!=E_OK){
```

Understanding Constants and Variables

Constants and variables are used in TSL to manipulate data. A constant is a value that never changes. It can be a number, character, or a string. A variable, in contrast, can change its value each time you run a test.

Variable and constant names can include letters, digits, and underscores (_). The first character must be a letter or an underscore. TSL is case sensitive; therefore, y and Y are two different characters. Certain words are reserved by TSL and may not be used as names.

You do not have to declare variables you use outside of function definitions in order to determine their type. If a variable is not declared, WinRunner determines its type (auto, static, public, extern) when the test is run.

For example, the following statement uses a variable to store text that appears in a text box.

```
edit_get_text ("Name:", text);
        report_msg ("The Customer Name is " & text);
```

WinRunner reads the value that appears in the File Name text box and stores it in the *text* variable. A **report_msg** statement is used to display the value of the text variable in a report. For more information, see "Sending Messages to the Test Results Window" on page 116. For information about variable and constant declarations, see Chapter 10, "Creating User-Defined Functions."

Performing Calculations

You can create tests that perform simple calculations using mathematical operators. For example, you can use a multiplication operator to multiply the values displayed in two text boxes in your application. TSL supports the following mathematical operators:

+	addition
-	subtraction
-	negation (a negative number - unary operator)
*	multiplication
/	division
%	modulus
^ or **	exponent
++	increment (adds 1 to its operand - unary operator)
--	decrement (subtracts 1 from its operand - unary operator)

TSL supports five additional types of operators: concatenation, relational, logical, conditional, and assignment. It also includes functions that can perform complex calculations such as **sin** and **exp**. See the *TSL Reference* for more information.

The following example uses the Flight Reservation application. WinRunner reads the price of both an economy ticket and a business ticket. It then checks whether the price difference is greater than $100.

```
# Select Economy button
set_window ("Flight Reservation");
button_set ("Economy", ON);

# Get Economy Class ticket price from price text box
edit_get_text ("Price:", economy_price);

# Click Business.
button_set ("Business", ON);

# Get Business Class ticket price from price box
edit_get_text ("Price:", business_price);

# Check whether price difference exceeds $100
if ((business_price - economy_price) > 100)
tl_step ("Price_check", 1, "Price difference is too large.");
```

Creating Stress Conditions

You can create stress conditions in test scripts that are designed to determine the limits of your application. You create stress conditions by defining a loop which executes a block of statements in the test script a specified number of times. TSL provides three statements that enable looping: *for*, *while*, and *do/while*. Note that you cannot define a constant within a loop.

For Loop

A *for* loop instructs WinRunner to execute one or more statements a specified number of times. It has the following syntax:

```
for ( [ expression1 ]; [ expression2 ]; [ expression3 ] )
    statement
```

First, *expression1* is executed. Next, *expression2* is evaluated. If *expression2* is true, *expression3* is executed. The cycle is repeated as long as *expression2* remains true. If *expression2* is false, the *for* statement terminates and execution passes to the first statement immediately following.

For example, the *for* loop below selects the file UI_TEST from the File Name list in the Open window. It selects this file five times and then stops.

```
set_window ("Open")
for (i=0; i<5; i++)
    list_select_item ("File Name:_1", "UI_TEST"); # Item Number 2
```

While Loop

A *while* loop executes a block of statements for as long as a specified condition is true. It has the following syntax:

```
while ( expression )
    statement ;
```

While *expression* is true, the statement is executed. The loop ends when the expression is false.

For example, the *while* statement below performs the same function as the *for* loop above.

```
set_window ("Open");
i=0;
while (i<5)
    {
    i++;
    list_select_item ("File Name:_1", "UI_TEST"); # Item Number 2
    }
```

Do/While Loop

A *do/while* loop executes a block of statements for as long as a specified condition is true. Unlike the *for* loop and *while* loop, a *do/while* loop tests the conditions at the end of the loop, not at the beginning. A *do/while* loop has the following syntax:

```
do
            statement
while (expression);
```

The statement is executed and then the *expression* is evaluated. If the expression is true, then the cycle is repeated. If the *expression* is false, the cycle is not repeated.

For example, the *do/while* statement below opens and closes the Order dialog box of Flight Reservation five times.

```
set_window ("Flight Reservation");
i=0;
do
    {
    menu_select_item ("File;Open Order...");
    set_window ("Open Order");
    button_press ("Cancel");
    i++;
    }
while (i<5);
```

Incorporating Decision-Making Statements

You can incorporate decision-making into your test scripts using *if/else* or *switch* statements.

If/Else Statement

An *if/else* statement executes a statement if a condition is true; otherwise, it executes another statement. It has the following syntax:

```
if ( expression )
    statement1;
[ else
    statement2; ]
```

expression is evaluated. If *expression* is true, *statement1* is executed. If *expression* is false, *statement2* is executed.

For example, the *if/else* statement below checks that the Flights button in the Flight Reservation window is enabled. It then sends the appropriate message to the report.

```
#Open a new order
set_window ("Flight Reservation_1");
menu_select_item ("File; New Order");

#Type in a date in the Date of Flight: box
edit_set_insert_pos ("Date of Flight:", 0, 0);
type ("120196");

#Type in a value in the Fly From: box
list_select_item ("Fly From:", "Portland");

#Type in a value in the Fly To: box
list_select_item ("Fly To:", "Denver");

#Check that the Flights button is enabled
button_get_state ("FLIGHT", value);
if (value != ON)
    report_msg ("The Flights button was successfully enabled");
else
    report_msg ("Flights button was not enabled. Check that values for
                Fly From and Fly To are valid");
```

Switch Statement

A *switch* statement enables WinRunner to make a decision based on an expression that can have more than two values. It has the following syntax:

```
switch (expression )
{
    case case_1:
        statements
    case case_2:
        statements
    case case_n:
        statements
default: statement(s)
}
```

The *switch* statement consecutively evaluates each case expression until one is found that equals the initial expression. If no case is equal to the expression, then the default statements are executed. The default statements are optional.

Note that the first time a case expression is found to be equal to the specified initial expression, no further case expressions are evaluated. However, all subsequent statements enumerated by these cases are executed, unless you use a *break* statement to pass execution to the first statement immediately following the *switch* statement.

The following test uses the Flight Reservation application. It randomly clicks either the First, Business or Economy Class button. Then it uses the appropriate GUI checkpoint to verify that the correct ticket price is displayed in the Price text box.

```
arr[1]="First";arr[2]="Business";arr[3]="Economy";
while(1)
{
    num=int(rand()*3)+1;

    # Click class button
    set_window ("Flight Reservation");
    button_set (arr[num], ON);

    # Check the ticket price for the selected button
    switch (num)
    {
        case 1: #First
        obj_check_gui("Price:", "list1.ckl", "gui1", 1);
        break;
        case 2: #Business
        obj_check_gui("Price:", "list2.ckl", "gui2", 1);
        break;
        case 3: #Economy
        obj_check_gui("Price:", "list3.ckl", "gui3", 1);
        }
    }
```

Sending Messages to the Test Results Window

You can define a message in your test script and have WinRunner send it to the test results window. To send a message to a test results window, add a **report_msg** statement to your test script. The function has the following syntax:

report_msg (*message* **);**

The *message* can be a string, a variable, or a combination of both.

In the following example, WinRunner gets the value of the label property in the Flight Reservation window and enters a statement in the test results containing the message and the label value.

```
win_get_info("Flight Reservation", "label", value);
report_msg("The label of the window is " & value);
```

Starting Applications from a Test Script

You can start an application from a WinRunner test script using the **invoke_application** function. For example, you can open the application being tested every time you start WinRunner by adding an **invoke_application** statement to a startup test. See Chapter 23, "Initializing Special Configurations," for more information on startup tests.

Tip: You can use the **Run** tab of the Test Properties dialog box to open an application at the beginning of a test run. For more information, refer to Chapter 22, "Setting Properties for a Single Test" in the *Mercury WinRunner Basic Features User's Guide*.
You can also use a **system** statement to start an application. For more information, refer to the *WinRunner TSL Reference Guide*.

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

invoke_application (*file, command_option, working_dir, show* **);**

The *file* designates the full path of the application to invoke. The *command_option* parameter designates the command line options to apply. The *work_dir* designates the working directory for the application and *show* specifies how the application's main window appears when open.

For example, the statement:

```
invoke_application("c:\\flight4a.exe", "", "", SW_MINIMIZED);
```

starts the Flight Reservation application and displays it as an icon.

Defining Test Steps

After you run a test, WinRunner displays the overall result of the test (pass/fail) in the Report form. To determine whether sections of a test pass or fail, add **tl_step** statements to the test script.

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

tl_step (*step_name, status, description* **);**

The *step_name* is the name of the test step. The *status* determines whether the step passed (0), or failed (any value except 0). The *description* describes the step.

For example, in the following test script segment, WinRunner reads text from Notepad. The **tl_step** function is used to determine whether the correct text is read.

```
win_get_text("Document - Notepad", text, 247, 309, 427, 329);
if (text=="100-Percent Compatible")
    tl_step("Verify Text", 0, "Correct text was found in Notepad");
else
    tl_step("Verify Text", 1,"Wrong text was found in Notepad");
```

When the test run is complete, you can view the test results in the WinRunner Report. The report displays a result (pass/fail) for each step you defined with **tl_step**.

Note that if you are using Quality Center to plan and design tests, you should use **tl_step** to create test steps in your automated test scripts. For more information, refer to the *Mercury Quality Center User's Guide*.

Comparing Two Files

WinRunner enables you to compare any two files during a test run and to view any differences between them using the **file_compare** function.

While creating a test, you insert a **file_compare** statement into your test script, indicating the files you want to check. When you run the test, WinRunner opens both files and compares them. If the files are not identical, or if they could not be opened, this is indicated in the test report. In the case of a file mismatch, you can view both of the files directly from the report and see the lines in the file that are different.

Suppose, for example, your application enables you to save files under a new name (Save As...). You could use file comparison to check whether the correct files are saved or whether particularly long files are truncated.

To compare two files during a test run, you program a **file_compare** statement at the appropriate location in the test script. This function has the following syntax:

file_compare (*file_1*, *file_2* [,*save_file*] **);**

The *file_1* and *file_2* parameters indicate the names of the files to be compared. If a file is not in the current test folder, then the full path must be given. The optional *save_file* parameter saves the name of a third file, which contains the differences between the first two files.

In the following example, WinRunner tests the Save As capabilities of the Notepad application. The test opens the *win.ini* file in Notepad and saves it under the name *win1.ini*. The **file_compare** function is then used to check whether one file is identical to the other and to store the differences file in the test directory.

```
# Open win.ini using WordPad.
system("write.exe c:\win\win.ini");
set_window("win.ini - WordPad",1);

# Save win.ini as win1.ini
menu_select_item("File;Save As...");
set_window("Save As");
edit_set("File Name:_0","c:\Win\win1.ini");
set_window("Save As", 10);
button_press("Save");

# Compare win.ini to win1.ini and save both files to "save".
file_compare("c:\\win\\win.ini","c:\\win\\win1.ini","save");
```

For information on viewing the results of file comparison, refer to Chapter 21, "Analyzing Test Results" in the *Mercury WinRunner Basic Features User's Guide*.

Checking the Syntax of your TSL Script

When WinRunner runs a test, it checks each script line for basic syntax errors, like incorrect syntax or missing elements in **If**, **While**, **Switch**, and **For** statements.

For example, WinRunner will stop and fail a test run if it finds one of the following:

```
# if statement without then
if()
report_msg("Bad If Structure");

#while statement without end condition
while(1
{
    report_msg("Bad While Structure");
}

#for statement without closing brackets
for(i=0;i<5;i++)
{
```

You can use the **Syntax Check** options to check for these types of syntax errors before running your test. You can run the syntax check from the beginning of your test or starting from a selected line in your test. This enables you to quickly check your test for syntax errors so that you can catch them without having to run the entire test.

To run a syntax check for your entire text, choose **Tools** > **Syntax Check** > **Syntax Check from Top**.

To run a syntax check from a selected point in your test, click a line in the left gutter to set the location of the arrow. Then choose **Tools** > **Syntax Check** > **Syntax Check from Arrow**.

Tip: If the left gutter is not visible, choose **Tools** > **Editor Options**, and select **Visible gutter** in the **Options** tab.

If a syntax error is found during the syntax check, a message box describes the error.

8

Generating Functions

Visual programming helps you add TSL statements to your test scripts quickly and easily.

This chapter describes:

- About Generating Functions
- Generating a Function for a GUI Object
- Selecting a Function from a List
- Assigning Argument Values
- Modifying the Default Function in a Category

About Generating Functions

When you record a test, WinRunner generates TSL statements in a test script each time you click a GUI object or use the keyboard. In addition to the recordable functions, TSL includes many functions that can increase the power and flexibility of your tests. You can easily add functions to your test scripts using WinRunner's visual programming tool, the Function Generator.

The Function Generator provides a quick, error-free way to program scripts. You can:

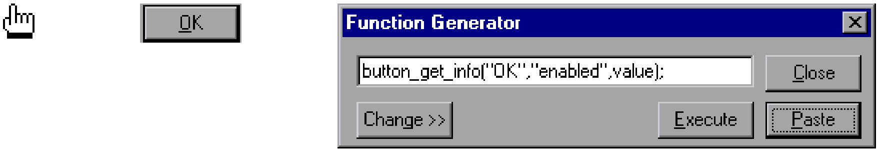
- Add Context Sensitive functions that perform operations on a GUI object or get information from the application being tested.
- Add Standard and Analog functions that perform non-Context Sensitive tasks such as synchronizing test execution or sending user-defined messages to a report.

- Add Customization functions that enable you to modify WinRunner to suit your testing environment.

You can add TSL statements to your test scripts using the Function Generator in two ways: by pointing to a GUI object, or by choosing a function from a list. When you choose the Insert Function command and point to a GUI object, WinRunner suggests an appropriate Context Sensitive function and assigns values to its arguments. You can accept this suggestion, modify the argument values, or choose a different function altogether.

By default, WinRunner suggests the default function for the object. In many cases, this is a **get** function or another function that retrieves information about the object. For example, if you choose **Insert** > **Function** > **For Object/Window** and then click an OK button, WinRunner opens the Function Generator dialog box and generates the following statement:

```
button_get_info("OK",”enabled”, value);
```



```
button_get_info("OK","enabled",value);
```

This statement examines the enabled property of the OK button and stores the current value of the property in the *value* variable. The *value* can be 1 (enabled), or 0 (disabled).

To select another function for the object, click **Change**. Once you have generated a statement, you can perform either or both of the following options:

- *Paste* the statement into your test script. When required, a **set_window** statement is inserted automatically into the script before the generated statement.
- *Execute* the statement from the Function Generator.

Note that if you point to an object that is not in the GUI map, the object is automatically added to the temporary GUI map file when the generated statement is executed or pasted into the test script.

Note: You can customize the Function Generator to include the user-defined functions that you most frequently use in your test scripts. You can add new functions and new categories and sub-categories to the Function Generator. You can also set the default function for a new category. For more information, see Chapter 22, “Customizing the Function Generator.” You can also change the default function for an existing category. For more information, see “Modifying the Default Function in a Category” on page 129.

Generating a Function for a GUI Object

With the Function Generator, you can generate a Context Sensitive function simply by pointing to a GUI object in your application. WinRunner examines the object, determines its class, and suggests an appropriate function. You can accept this default function or select another function from a list.

Using the Default Function for a GUI Object

When you generate a function by pointing to a GUI object in your application, WinRunner determines the class of the object and suggests a function. For most classes, the default function is a **get** function. For example, if you click a list, WinRunner suggests the **list_get_selected** function.

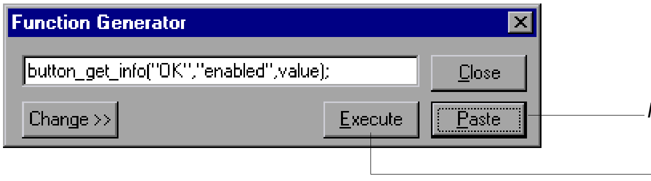
To use the default function for a GUI object:

1. Choose **Insert** > **Function** > **For Object/Window** or click the **Insert Function for Object/Window** button on the User toolbar. WinRunner shrinks to an icon and the mouse pointer becomes a pointing hand.
2. Point to a GUI object in the application being tested. Each object flashes as you pass the mouse pointer over it.
3. Click an object with the left mouse button. The Function Generator dialog box opens and shows the default function for the selected object. WinRunner automatically assigns argument values to the function.

To cancel the operation without selecting an object, click the right mouse button.

4 To *paste* the statement into the script, click **Paste**. The function is pasted into the test script at the insertion point and the Function Generator dialog box closes.

To *execute* the function, click **Execute**. The function is executed. Clicking Execute does not paste the statement into the script.



```
button_get_info("OK","enabled",value);
```

Pastes the function into the script

Only executes the function

5 Click **Close** to close the dialog box.

Selecting a Non-Default Function for a GUI Object

If you do not want to use the default function suggested by WinRunner, you can choose a different function from a list.

To select a non-default function for a GUI object:

1 Choose **Insert** > **Function** > **For Object/Window** or click the **Insert Function for Object/Window** button on the User toolbar. WinRunner is minimized and the mouse pointer becomes a pointing hand.

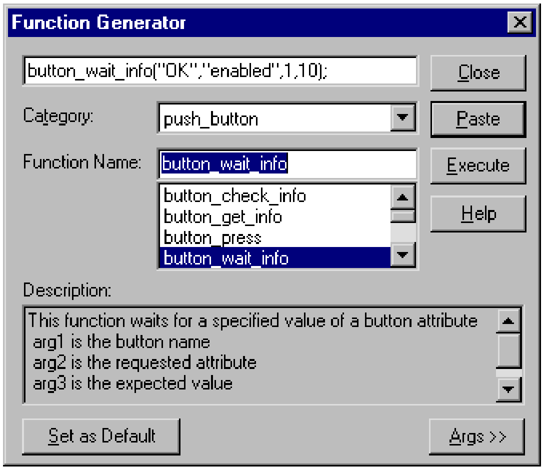
2 Point to a GUI object in the application being tested. Each object flashes as you pass the mouse pointer over it.

3 Click an object with the left mouse button. The **Function Generator** dialog box opens and displays the default function for the selected object. WinRunner automatically assigns argument values to the function.

To cancel the operation without selecting an object, click the right mouse button.

4 In the Function Generator dialog box, click **Change**. The dialog box expands and displays a list of functions. The list includes only functions that can be used on the GUI object you selected. For example, if you select a push button, the list displays **button_get_info**, **button_press**, etc.

5 In the **Function Name** list, select a function. The generated statement appears at the top of the dialog box. Note that WinRunner automatically fills in argument values. A description of the function appears at the bottom of the dialog box.



button_wait_info("OK","enabled",1,10);

Category: push_button

Function Name: button_wait_info

button_check_info
button_get_info
button_press
button_wait_info

Description:

This function waits for a specified value of a button attribute
arg1 is the button name
arg2 is the requested attribute
arg3 is the expected value

6 If you want to modify the argument values, click **Args**. The dialog box expands and displays a text box for each argument. See “Assigning Argument Values” on page 127 for more information on how to fill in the argument text boxes.

7 To *paste* the statement into the test script, click **Paste**. The function is pasted into the test script at the insertion point.

To *execute* the function, click **Execute**. The function is immediately executed but is not pasted into the test script.

8 You can continue to generate function statements for the same object by repeating the steps above without closing the dialog box. The object you selected remains the active object and arguments are filled in automatically for any function selected.

9 Click **Close** to close the dialog box.

Selecting a Function from a List

When programming a test, perhaps you know the task you want the test to perform but not the exact function to use. The Function Generator helps you to quickly locate the function you need and insert it into your test script. Functions are organized by category; you select the appropriate category and the function you need from a list. A description of the function is displayed along with its parameters.

To select a function from a list:

1. Choose **Insert** > **Function** > **From Function Generator** or click the **Insert Function from Function Generator** button on the User toolbar to open the Function Generator dialog box.

2. In the **Category** list, select a function category. For example, if you want to view menu functions, select menu. If you do not know which category you need, use the default *all_functions*, which displays all the functions listed in alphabetical order.

3. In the **Function Name** list, choose a function. If you select a category, only the functions that belong to the category are displayed in the list. The generated statement appears at the top of the dialog box. Note that WinRunner automatically fills in the default argument values. A description of the function appears at the bottom of the dialog box.

4. To define or modify the argument values, click **Args**. The dialog box expands and displays a text box for each argument. See “Assigning Argument Values” on page 127 to learn how to fill in the argument text boxes.

5. To *paste* the statement into the test script, click **Paste**. The function is pasted into the test script at the insertion point.

   To *execute* the function, click **Execute**. The function is immediately executed but is not pasted into the test script.

6. You can continue to generate additional function statements by repeating the steps above without closing the dialog box.

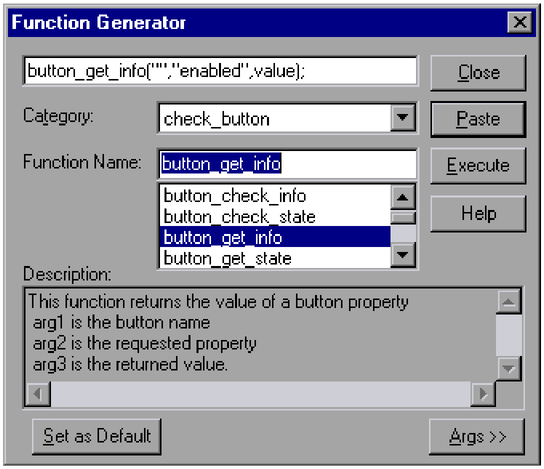
7. Click **Close** to close the dialog box.

Assigning Argument Values

When you generate a function using the Function Generator, WinRunner automatically assigns values to the function's arguments. If you generate a function by clicking a GUI object, WinRunner evaluates the object and assigns the appropriate argument values. If you choose a function from a list, WinRunner fills in default values where possible, and you type in the rest.

To assign or modify argument values for a generated function:

1. Choose **Insert** > **Function** > **From Function Generator** or click the **Insert Function from Function Generator** button on the User toolbar to open the Function Generator dialog box.



Function Generator

button_get_info("","enabled",value);

Category: check_button

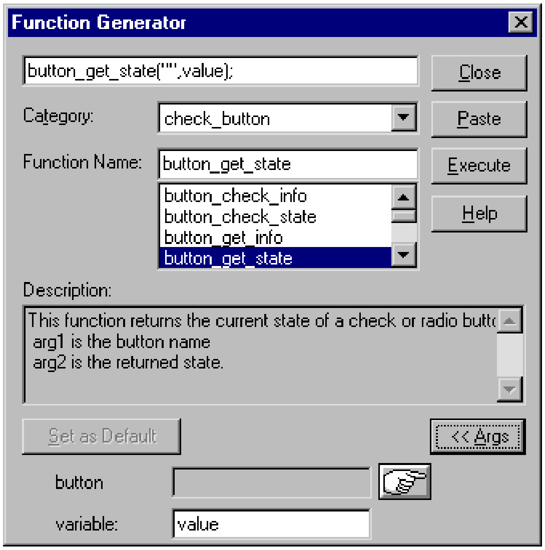
Function Name: button_get_info

button_check_info
button_check_state
button_get_info
button_get_state

Description:
This function returns the value of a button property
arg1 is the button name
arg2 is the requested property
arg3 is the returned value.

2. In the **Category** list, select a function category. For example, if you want to view menu functions, select menu. If you do not know which category you need, use the default *all_functions*, which displays all the functions listed in alphabetical order.
3. In the **Function Name** list, choose a function. If you select a category, only the functions that belong to the category are displayed in the list. The generated statement appears at the top of the dialog box. Note that WinRunner automatically fills in the default argument values. A description of the function appears at the bottom of the dialog box.

4 Click **Args**. The dialog box displays the arguments in the function.



Function Generator

button_get_state("",value);

Category: check_button

Function Name: button_get_state

button_check_info
button_check_state
button_get_info
button_get_state

Description:
This function returns the current state of a check or radio butto
arg1 is the button name
arg2 is the returned state.

button

variable: value

5 Assign values to the arguments. You can assign a value either manually or automatically.

To *manually* assign values, type in a value in the argument text box. For some text boxes, you can choose a value from a list.

To *automatically* assign values, click the pointing hand and then click an object in your application. The appropriate values appear in the argument text boxes.

Note that if you click an object that is not compatible with the selected function, a message informs you that the function is not applicable for the object you selected. Click **OK** to clear the message and return to the Function Generator.

Modifying the Default Function in a Category

In the Function Generator, each function category has a default function. When you generate a function by clicking an object in your application, WinRunner determines the appropriate category for the object and suggests the default function. For most Context Sensitive function categories, this is a **get** function. For example, if you click a text box, the default function is **edit_get_text**. For Analog, Standard and Customization function categories, the default is the most commonly used function in the category. For example, the default function for the system category is **invoke_application**.

If you find that you frequently use a function other than the default for the category, you can make it the default function.

To change the default function in a category:

1. Choose **Insert** > **Function** > **From Function Generator** or click the **Insert Function from Function Generator** button on the User toolbar to open the Function Generator dialog box.
2. In the **Category** list, select a function category. For example, if you want to view menu functions, select menu.
3. In the **Function Name** list, select the function that you want to make the default.
4. Click **Set as Default**.
5. Click **Close**.

The selected function remains the default function in its category until it is changed or until you end your WinRunner session.

To permanently save changes to the default function setting, add a **generator_set_default_function** statement to your startup test. For more information on startup tests, see Chapter 23, “Initializing Special Configurations.”

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

generator_set_default_function (*category_name*, *function_name* **);**

For example:

```
generator_set_default_function ("push_button", "button_press");
```

sets **button_press** as the default function for the push_button category.

9

Calling Tests

The tests you create with WinRunner can call, or be called by, any other test. When WinRunner calls a test, parameter values can be passed from the calling test to the called test.

This chapter describes:

- About Calling Tests
- Using the Call Statement
- Returning to the Calling Test
- Setting the Search Path
- Working with Test Parameters
- Viewing the Call Chain

About Calling Tests

By adding **call** statements to test scripts, you can create a modular test tree structure containing an entire test suite. A modular test tree consists of a main test that calls other tests and controls test execution.

When WinRunner interprets a **call** statement in a test script, it opens and runs the called test. Input parameter values may be passed to this test from the calling test. When the called test is completed, WinRunner returns to the calling test and continues the test run. If the called test returned output parameter values to the calling test, the calling test can use those parameters in its subsequent steps. Note that a called test may also call other tests.

By adding decision-making statements to the test script, you can use a main test to determine the conditions that enable a called test to run.

For example:

```
rc= call login ("Jonathan", "Mercury");
if (rc == E_OK)
{
    call insert_order();
}
else
{
    tl_step ("Call Login", 1, "Login test failed");
    call open_order ();
}
```

This test calls the login test. If login is executed successfully, WinRunner calls the insert_order test. If the login test fails, the open_order test runs.

Called tests can have parameterized values. There are two types of parameters:

- **Input—**The called test receives parameters from the calling test and uses them to replace data in the test.
- **Output—**The called test returns parameters to the calling test, which can then use the parameters' data.

You commonly use **call** statements in a batch test. A batch test allows you to call a group of tests and run them unattended. It suppresses messages that are usually displayed during execution, such as one reporting a bitmap mismatch. For more information, see Chapter 14, "Running Batch Tests."

Note: If a called test that was created in the *GUI Map File per Test* mode references GUI objects, it may not run properly in the *Global GUI Map File* mode.

At each break during a test run—such as after a Step command, at a breakpoint, or at the end of a test, you can view the current chain of called tests and functions in the Call Chain pane of the Debug Viewer window. You can also click the **Display Test** button in the Call Chain pane to display the test that is currently running.

You can also use the Insert Call to QuickTest dialog box or insert a **call_ex** statement to call a QuickTest test. For more information, see Chapter 25, "Integrating with QuickTest Professional."

Using the Call Statement

You can use two types of call statements to invoke one test from another:

- A **call** statement invokes a test from within another test.
- A **call_close** statement invokes a test from within another test and closes the test when the test is completed.

The **call** statement has the following syntax:

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

The **call_close** statement has the following syntax:

call_close *test_name* **(** [*parameter*$_1$, *parameter*$_2$, ... *parameter*$_n$] **);**

The *test_name* is the name of the test to invoke. The *parameters* are the input and/or output parameters defined for the called test. For more information on using parameters, see "Guidelines for Working with Test Parameters" on page 139.

Any called test must be stored in a folder specified in the search path, or else be called with the full pathname enclosed within quotation marks.

For example:

```
call "w:\\tests\\my_test" ();
```

While running a called test, you can pause execution and view the current call chain. For more information, see "Viewing the Call Chain" on page 145.

Returning to the Calling Test

The **treturn** and **texit** statements are used to stop execution of called tests and return a value to the call statement

The value of the **treturn** or **texit** statement in the called test acts as the return value of the entire call statement in the calling test. You can return additional values to the calling test using output parameters. For more information, see “Working with Test Parameters” on page 137.

- The **treturn** statement stops the current test and returns control to the calling test.
- The **texit** statement stops test execution entirely, unless tests are being called from a batch test. In this case, control is returned to the main batch test.

Both functions provide a return value for the called test.

treturn

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

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

The optional *expression* is the value returned to the **call** statement used to invoke the test.

For example:

```
# test_a
if (call test_b() == "success")
    report_msg("test_b succeeded");

# test_b
if (win_check_bitmap ("Paintbrush - SQUARES.BMP", "Img_2", 1))
    treturn("success");
else
    treturn("failure");
```

In the above example, test_a calls test_b. If the bitmap comparison in test_b is successful, then the string “success” is returned to the calling test, test_a. If there is a mismatch, then test_b returns the string “failure” to test_a.

texit

When tests are run interactively, the **texit** statement discontinues test execution. However, when tests are called from a batch test, **texit** ends execution of the current test only; control is then returned to the calling batch test. The syntax is:

texit [(*expression* **)];**

The optional *expression* is the value returned to the call statement that invokes the test.

For example:

```
# batch_test
return_val = call help_test();
report_msg("help returned the value" return_val);

# help_test
call select_menu(help, index);
msg = get_text(4,30,12,100);
if (msg == "Index help is not yet implemented")
    texit("index failure");
...
```

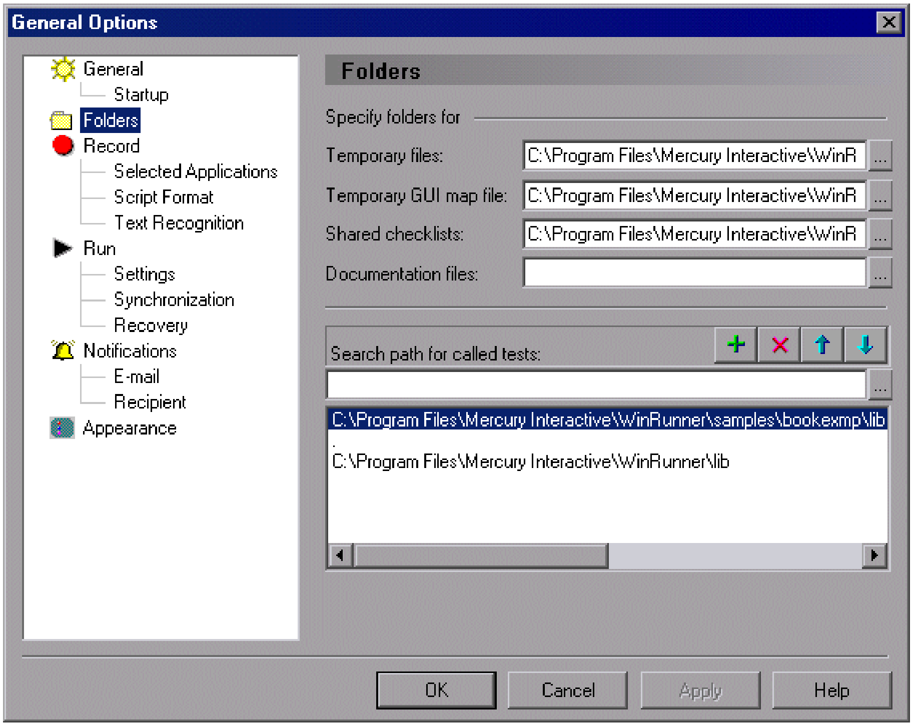
In the above example, batch_test calls help_test. In help_test, if a particular message appears on the screen, execution is stopped and control is returned to the batch test. Note that the return value of help_test is also returned to the batch test, and is assigned to the variable *return_val*.

For more information on batch tests, see Chapter 14, “Running Batch Tests.”

Setting the Search Path

The search path determines the directories that WinRunner will search for a called test.

To set the search path, choose **Tools > General Options**. The General Options dialog box opens. Click the **Folders** category and choose a search path in the **Search path for called tests** box. WinRunner searches the directories in the order in which they are listed in the box. Note that the search paths you define remain active in future testing sessions.



Temporary files: C:\Program Files\Mercury Interactive\WinR

Temporary GUI map file: C:\Program Files\Mercury Interactive\WinR

Shared checklists: C:\Program Files\Mercury Interactive\WinR

Documentation files:

Search path for called tests:

```
C:\Program Files\Mercury Interactive\WinRunner\samples\bookexmp\lib
.
C:\Program Files\Mercury Interactive\WinRunner\lib
```

- ✚ ➤ To add a folder to the search path, type in the folder name in the text box and click the **Add** button.
- ↑ ↓ ➤ Use the **Up** and **Down** buttons to position this folder in the list.
- ✕ ➤ To delete a search path, select its name from the list and click the **Delete** button.

For more information about how to set a search path in the General Options dialog box, refer to Chapter 23, "Setting Global Testing Options" in the *Mercury WinRunner Basic Features User's Guide*.

You can also set a search path by adding a **setvar** statement to a test script. A search path set using **setvar** is valid for all tests in the current session only.

For example:

```
setvar ("searchpath", "<c:\\ui_tests>");
```

This statement tells WinRunner to search the *c:\ui_tests* folder for called tests. For more information on using the **setvar** function, see Chapter 21, "Setting Testing Options from a Test Script."

Note: If WinRunner is connected to Quality Center, you can also set a search path within a Quality Center database. For more information, see "Using TSL Functions with Quality Center" on page 407.

Working with Test Parameters

When a test calls another test, it can supply the called test with one or more parameters.

A WinRunner test can receive data in input parameters and return values in output parameters, much like a TSL function. The calling test supplies values for these input and output parameters as arguments in the call statement. When working with a call chain, you can use parameters to pass data from one test to another.

Note: You can run a test that has input parameters defined in it, without using another test to call it. This is particularly useful for debugging a test before placing it in a call chain. You give such a test the values for its input parameters when you run it. For more information, refer to Chapter 20, "Understanding Test Runs" in the *Mercury WinRunner Basic Features User's Guide*.

Understanding Parameter Types

There are two types of test parameters: input and output. You define both types in the test you want to call, and you initialize them by entering them as arguments in the test containing the call statement to that test.

You can define any number of input and output parameters for a test. You define the parameters that a test can receive in the Parameters tab of the Test Properties dialog box. For more information on defining test parameters, refer to Chapter 22, "Setting Properties for a Single Test" in the *Mercury WinRunner Basic Features User's Guide*.

- An input parameter is a variable that is assigned a value from outside the called test. You can define one or more input parameters for a test; any calling test can then supply values for these parameters. These values can be the values themselves, or variables that contain the values. If the calling test does not supply values then the default values, if defined, are used.

  For example, suppose you define two input parameters, *starting_x* and *starting_y* for a test. The purpose of these parameters is to assign a value to the initial mouse pointer position when the test is called. The two values supplied by a calling test will supply the x- and y-coordinates of the mouse pointer.

- An output parameter is a variable whose value is generated within the called test (by calculation or by retrieving values during the test), and this value is then returned to the calling test. The calling test initializes each output parameter by including it as an argument in the call statement. After the called test runs and returns the output parameter values, the calling test can use those values by referring to the arguments it used in its test call.

For example, a test reads information from two edit boxes. You define two output parameters in the test. Steps in the test assign data to each of them such as data retrieved from two edit boxes. You then call this test from another test, and include two variables as arguments in the test call, *First_Name* and *Last_Name*, that correspond to the two output parameters of the called test. After the called test runs, the calling test can refer to *First_Name* and *Last_Name* in its script, and will use the values returned by the called test.

Guidelines for Working with Test Parameters

When working with test parameters consider the following:

- In test calls, you must supply all input parameters before any output parameters.
- If no parameters are defined for the called test, the **call** statement must contain an empty set of parentheses.
- If you do not supply a value for a defined input parameter and a default value has been defined in the called test, the default value is used. Otherwise the parameter is treated as an empty value.
- If you do not supply a variable for a defined output parameter, then the retrieved parameter value is not returned to the calling test.
- If you pass more parameters to a called test than the number of parameters that are actually defined in that test, then during the test run a warning message ("Warning: Test <path to test>: too many arguments") is displayed.
- Output parameters are supported only when working with WinRunner call chains. When working with QuickTest Professional or Quality Center, you should not call WinRunner tests containing output parameters.
- Parameters sent as arrays must subsequently be handled as arrays in the script, in both called and calling tests. Similarly, parameters sent as non-array variables cannot be subsequently handled as arrays.
- It is recommended to add _IN and _OUT as suffixes (or IN_ and OUT_ as prefixes) for the parameters you define. There prefixes or suffixes make your test easier to read.

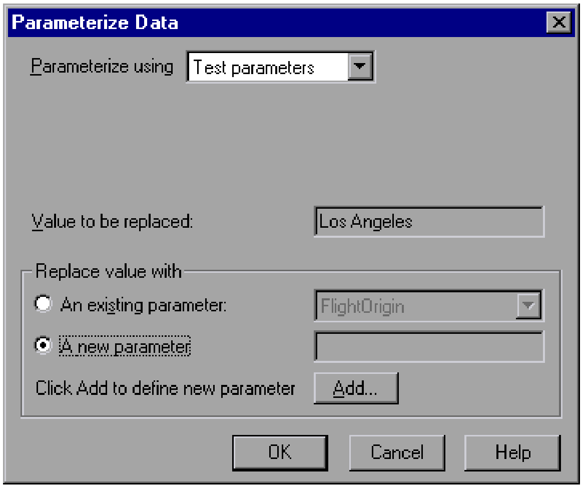
Defining Test Parameters

You can define test parameters in the **Parameters** tab of the Test Properties dialog box or in the Parameterize Data dialog box.

- Use the **Parameters** tab of the Test Properties dialog box when you want to manage the parameters of the test including adding, modifying, and deleting the parameters list for the test. For more information about **Parameters** tab of the Test Properties dialog box, refer to Chapter 22, "Setting Properties for a Single Test" in the *Mercury WinRunner Basic Features User's Guide*.
- Use the Parameterize Data dialog box when you want to replace existing data from the test with input parameters. You can replace the data with existing input parameters or create new ones.

To define test input parameters in the Parameterize Data dialog box:

1. In your test script, select the data that you want to parameterize.
2. Choose **Table** > **Parameterize Data** or right-click the selected data and choose **Parameterize Data**. The Parameterize Data dialog box opens.
3. In the **Parameterize using** box, select **Test parameters**.



Parameterize Data

Parameterize using: Test parameters

Value to be replaced: Los Angeles

Replace value with

○ An existing parameter: FlightOrigin

● A new parameter

Click Add to define new parameter

4. In the **Replace value with** box, select **An existing parameter** or **A new parameter.**

5 Select the parameter you want to use to replace the selected value.

- ➤ If you selected **An existing parameter** in step 4, select the parameter you want to use from the list. Note that the parameters listed here are the same as those listed in the **Parameters** tab of the Test Properties dialog box.
- ➤ If you selected **A new parameter** in step 4, click the **Add** button. The Parameter Properties dialog box opens. Add a new parameter as described in Chapter 22, "Setting Properties for a Single Test" in the *Mercury WinRunner Basic Features User's Guide*. The new parameter is displayed in the new parameter field. The new parameter is also added to the parameters list in the **Parameters** tab of the Test Properties dialog box.

6 Click **OK**.

The data selected in the test script is replaced with the input parameter you created or selected.

7 Repeat steps 1 to 6 for each argument you want to parameterize.

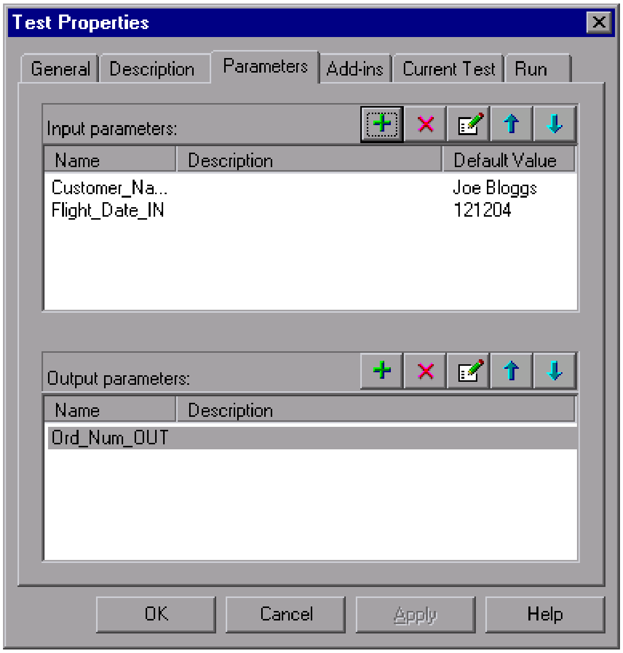
Using Test Parameters—An Example

In the example below, the calling test checks whether a certain customer is entitled to a special discount. To retrieve the customer's order number, it calls another test, whose task is to return an order number based upon supplied customer and flight date data.

```
# calling test
    Cust_Name = “Joe Bloggs”
    Flt_Date = “12122004”
    call “C:\\WinRunner Tests\\Get_Ord_Num”(Cust_Name, Flt_Date,
    Order_Number);
    if (Order_Number%50==0)
        report_msg(“Prizewinner Discount!”);
    else
        report_msg(“Regular ticket”);

#called test, name Get_Ord_Num
# Flight Reservation
    win_activate ("Flight Reservation");
    set_window ("Flight Reservation", 1);
    wait(1);
    menu_select_item ("File;Open Order...");
# Open Order
    set_window ("Open Order", 1);
    button_set ("Customer Name", ON);
    edit_set ("Edit", "Customer_Name_IN");
    button_set ("Flight Date", ON);
    obj_type ("MSMaskWndClass","Flight_Date_IN");
    wait(1);
    button_press ("OK");
# Search Results
    set_window ("Search Results", 1);
    button_press ("OK");
# Flight Reservation
    set_window ("Flight Reservation", 1);
    win_activate ("Flight Reservation");
    edit_get_text(“Order No:”Ord_Num_OUT);
```

Three parameters are defined in the called test: Customer_Name_IN, Flight_Date_IN, and Order_No_OUT. Note that these parameter names clearly show the parameter type.



Input parameters:

Name	Description	Default Value
Customer_Na...		Joe Bloggs
Flight_Date_IN		121204

Output parameters:

Name	Description
Ord_Num_OUT	

The calling test supplies the customer name and flight date as input parameters. The called test uses the customer name and flight date input parameters in a search for the corresponding order number. The retrieved number is returned as the output parameter.

The calling test then uses this output parameter in its own script to see if this value is divisible by 50, and on that basis determines whether the customer is a prize winner.

Test Parameter Scope

The parameter defined in the called test 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 test. This means that a copy is passed to the called test. This copy is local; if its value is changed in the called test, the original value in the calling test is not affected. For example:

```
# test_1 (calling_test)
i = 5;
call test_2(i);
pause(i); # Opens a message box with the number "5" in it
# test_2 (called test_1), with formal parameter x
x = 8;
pause(x); # Opens a message box with the number "8" in it
```

In the calling test (test_1), the variable *i* is assigned the value 5. This value is passed to the called test (test_2) as the value for the formal parameter x. Note that when a new value (8) is assigned to x in test_2, this change does not affect the value of *i* in test_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 test affects the corresponding array in the calling test. For example:

```
# test_q
a[1] = 17;
call test_r(a);
pause(a[1]); # Opens a message box with the number "104" in it
# test_r, with parameter x
x[1] = 104;
```

In the calling test (test_q), element 1 of array *a* is assigned the value 17. Array *a* is then passed to the called test (test_r), which has a formal parameter *x*. In test_r, the first element of array *x* is assigned the value 104.

Unlike the previous example, this change to the parameter in the called test does affect the value of the parameter in the calling test, because the parameter is an array.

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

The test segments below illustrates the use of global variables. Note that test_a is not called, but is run directly.

```
# test_a, with input parameter k
# Note that the ampersand (&) is a bitwise AND operator. It signifies
concatenation.
i = 1;
j = 2;
k = 3;
call test_b(i);
pause(j & k & l); # Opens a message box with the number '256' in it
# test_b, with input parameter j
# Note that the ampersand (&) is a bitwise AND operator. It signifies
concatenation.
j = 4;
k = 5;
l = 6;
pause(j & k & l); # Opens a message box with the number '456' in it
```

Viewing the Call Chain

At each break during a test run—such as after a Step command, at a breakpoint, or at the end of a test, you can view the current chain of called tests and functions in the Call Chain pane of the Debug Viewer window.

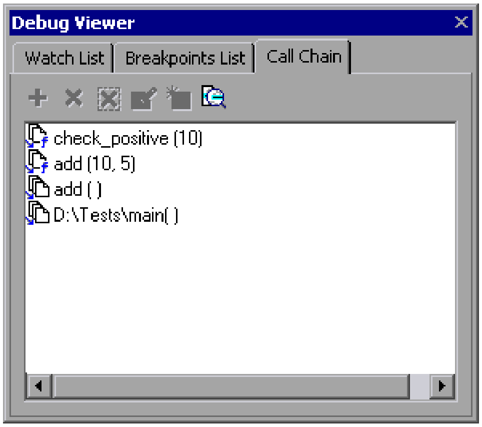
To view the current call chain:

1 If the Debug Viewer window is not currently displayed, or the Call Chain pane is not open in the window, choose **Debug** > **Call Chain** to display it. If the Call Chain pane is open, but a different pane is currently displayed, click the **Call Chain** tab to display it.

2 Ensure that your called tests have breakpoints in places where you would like to view the call chain. Alternatively, use the Step commands to control the run of the test.

For more information on Step commands, see Chapter 16, “Controlling Your Test Run.”

3 When the test pauses, view the call chain in the Call Chain pane of the Debug Viewer.



check_positive (10)
add (10, 5)
add ( )
D:\Tests\main( )

Tip: The Debug Viewer window can be displayed as a docked window within the WinRunner window, or it can be a floating window that you can drag to any location on your screen.

---

4 To view the script of a test in the call chain, double-click a test or function, or select the test or function in the list and click **Display test**. The selected test or function becomes the active window in WinRunner.

10

Creating User-Defined Functions

You can expand WinRunner's testing capabilities by creating your own TSL functions. You can use these user-defined functions in a test or a compiled module. This chapter describes:

➤ About Creating User-Defined Functions

➤ Function Syntax

➤ Return and Exit Statements

➤ Variable, Constant, and Array Declarations

➤ Example of a User-Defined Function

About Creating User-Defined Functions

In addition to providing built-in functions, TSL allows you to design and implement your own functions.

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

You can use functions that you create in the test in which they reside, or you can store them in a compiled module for use in other tests. For more information about compiled modules, see "Understanding the Contents of a Compiled Module" on page 159.

For example, you could create a function called open_flight that loads a GUI map file, starts the Flight Reservation application, and logs into the system, or resets the main window if the application is already open.

A function can be called from anywhere in a test script. Since it is already compiled, execution time is accelerated. For instance, suppose you create a test that opens a number of files and checks their contents. Instead of recording or programming the sequence that opens the file several times, you can 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;
}

Function Classes

The class of a function can be either *static*, *public* or *external*.

A *static* function is available only to the test or module within which the function is defined.

Once you execute a *public* function, it is available to all tests, for as long as the test containing the function remains open (until you manually click the **Stop** button). This is convenient when you want the function to be accessible from called tests. However, if you want to create a function that will be available to many tests, you should place it in a compiled module. Once you have loaded a compiled module, its functions are available for all tests until you unload it. For more information on loading and unloading a compiled module, see “Loading a Function or Compiled Module” on page 163 and “Loading and Unloading a Compiled Module” on page 170.

An *external* function behaves like a public function, except that while its declaration is in the local test or compiled module, its implementation code resides in an external source. The most common example is a function that is defined in a DLL. You must load the DLL in which the function is defined.

You can then declare the function in a test or compiled module, and then load it. Once it is loaded, your tests can call it. For more information, refer to the chapter "Calling Functions from External Libraries."

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

Function Parameters

Parameters need not be explicitly declared. They can be of mode *in*, *out*, or *inout*. For all non-array parameters, the default mode is *in*. For array parameters, the default is *inout*. 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 or inside the function.

A parameter designated as out or inout must be a variable name, not an expression. When you call a function containing an *out* or an *inout* parameter, the argument corresponding to that parameter must be a variable, and not an expression. For example, consider a user-defined function with the following syntax:

```
function get_date (out todays_date) { ... }
```

Proper usage of the function call would be:

```
get_date (todays_date);
```

Conversely, the following function calls contain expressions and are therefore illegal:

get_date (date[i]); **or** get_date ("Today's date is"& todays_date);

Array parameters are designated by square brackets. For example, the following parameter list in a user-defined function indicates that variable *a* is an array:

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

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

Note: You can define up to 15 parameters in a user-defined function.

Return and Exit Statements

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

return ([*expression*] **);**

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

The texit statement can be used to stop a function or test run. The syntax is:

texit ([*expression*]);

When a test is run interactively, texit discontinues the test run entirely. When a test is run in batch mode, the statement ends execution of the current main test only; control is then returned to the calling batch test. The texit function also returns the value of the evaluated expression to the calling function or test.

Note: QuickTest does not support **texit** statements inside called functions. If QuickTest calls a WinRunner function containing a **texit** statement, the function call fails.

Variable, Constant, and Array Declarations

Declaration is usually optional in TSL. In functions, however, variables, constants, and arrays must all be declared. The declaration can be within the function itself, or anywhere else within the test script or compiled module containing the function. You can find additional information about declarations in the *TSL Reference*.

Declaring 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 *class* defines the scope of the variable. It can be one of the following:

auto—An auto variable can be declared only within a function and is local to that function. It exists only for as long as the function is running. A new copy of the variable is created each time the function is called.

static—A static variable is local to the function, test, or compiled module in which it is declared. The variable retains its value until the test is terminated by an Abort command. This variable is initialized each time the definition of the function is executed.

Note: In compiled modules, a **static** variable is initialized whenever the compiled module is compiled.

public—A public variable can be declared only within a test or module, and is available for all functions, tests, and compiled modules.

extern—An extern declaration indicates a reference to a public variable declared outside of the current test or module.

Remember that you must declare all variables used in a function within the function itself, or within the test or module that contains the function. If you wish to use a public variable that is declared outside of the relevant test or module, you must declare it again as **extern**.

The **extern** declaration must appear within a test or module, before the function code. An extern declaration cannot initialize a variable.

For example, suppose that in Test 1 you declare a variable as follows:

```
public window_color=green;
```

In Test 2, you write a user-defined function that accesses the variable window_color. Within the test or module containing the function, you declare window_color as follows:

```
extern window_color;
```

With the exception of the **auto** variable, all variables continue to exist until the Stop command is executed.

Note: In compiled modules, all variables continue to exist until the Stop command is executed with the exception of the **auto** and **public** variables. (The **auto** variables exist only as long as the function is running; **public** variables exist until exiting WinRunner.)

The following table summarizes the scope, lifetime, and availability (where the declaration can appear) of each type of variable:

Declaration	Scope	Lifetime	Declare the Variable in...
auto	local	end of function	function
static	local	until abort	function, test, or module
public	global	until abort	test or module
extern	global	until abort	function, test, or module

Note: In compiled modules, the Stop command initializes **static** and **public** variables. For more information, see Chapter 11, “Employing User-Defined Functions in Tests.”

Declaring 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 WinRunner.

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.)

Declaring 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 used for variable declarations (auto, static, public, extern).

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 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.

For example:

```
static gui_item [ ]={
    "class"="push_button",
    "label"="OK",
    "X_class"="XmPushButtonGadget",
    "X"=10,
    "Y"=60
    };
```

creates the following array elements:

```
gui_item ["class"]="push_button"
gui_item ["label"]="OK"
gui_item ["X_class"]="XmPushButtonGadget"
gui_item ["X"]=10
gui_item ["Y"]=60
```

Note that arrays are initialized once, the first time a function is run. If you edit the array's initialization values, the new values will not be reflected in subsequent test runs. To reset the array with the new initialization values, either interrupt test execution with the Stop command, or define the new array elements explicitly. For example:

Regular Initialization	**Explicit Definitions**

public number_list[] = {1,2,3};	number_list[0] = 1;
	number_list[1] = 2;
	number_list[2] = 3;

Statements

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

Example of a User-Defined Function

The following user-defined function opens the specified text file in an editor. It assumes that the necessary GUI map file is loaded. The function verifies that the file was actually opened by comparing the name of the file with the label that appears in the window title bar after the operation is completed.

```
function open_file (file)
{
    auto lbl;
    set_window ("Editor");

    # Open the Open form
    menu_select_item ("File;Open...");

    # Insert file name in the proper field and click OK to confirm
    set_window ("Open");
    edit_set(“Open Edit”, file);
    button_press ("OK");

    # Read window banner label
    win_get_info("Editor","label",lbl);

    #Compare label to file name
    if ( file != lbl)
        return 1;
    else
        return 0;
}
rc=open_file("c:\\dash\\readme.tx");
pause(rc);
```

11

Employing User-Defined Functions in Tests

You can call user-defined functions from within the test in which you defined them, from other tests containing loaded functions, and from loaded compiled modules.

This chapter describes:

- About Employing User-Defined Functions
- Understanding the Contents of a Compiled Module
- Using the Function Viewer
- Employing Functions Defined In Tests
- Employing Functions Defined in Compiled Modules

About Employing User-Defined Functions

You can employ user-defined functions in one of three ways:

- You can call a function from within the test in which you defined it. Your function call can include input and output arguments.

For example, the following simple function

```
public function Add6(x)
{
      return(x+6);
}
```

can be called with the following command:

```
y=Add6(Ord_Num);
```

- You can call a non-static function defined in any test that you have run. When you run a test, any public functions it contains are loaded and are available to any other test until you click the **Stop** button. When you click the **Stop** button, the loaded functions from all tests that are not compiled modules become unloaded. For more information, see "Employing Functions Defined In Tests" on page 167.
- You can call a non-static function from a loaded compiled module. A compiled module is a special test type that contains a library of functions that you may want to use often. You can load a compiled module using the Function Viewer or from a test script. For more information on the Function Viewer, see "Using the Function Viewer" on page 161. For more information on loading a compiled module from a test script, see "Employing Functions Defined in Compiled Modules" on page 168.

Note: Only public and external functions can be called using the last two of the above options. For more information, see "Function Classes" on page 148.

The remainder of this chapter discusses the contents of a compiled module, the Function Viewer and the last two of the above options.

Understanding the Contents of a Compiled Module

A compiled module, like a regular test you create in TSL, can be opened, edited, and saved. You indicate that a test is a compiled module in the **General** tab of the Test Properties dialog box, by selecting **Compiled Module** in the **Test Type** box. For more information, see "Creating a Compiled Module" on page 160.

The content of a compiled module differs from that of an ordinary test. For example, it cannot include checkpoints or any analog input such as mouse tracking. The purpose of a compiled module is not to perform a test, but to store the user-defined functions you use most frequently so that they can be quickly and conveniently accessed from other tests.

Unlike an ordinary test, all data objects (variables, constants, arrays) in a compiled module 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, see Chapter 10, "Creating User-Defined Functions."
- prototypes of external functions. For more information, see Chapter 12, "Calling Functions from External Libraries."
- **load** statements to other modules. For more information, see "Loading and Unloading a Compiled Module" on page 170.

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

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

Note: If you make changes to a function in a loaded compiled module, you must unload and reload the compiled module in order for the changes to take effect.

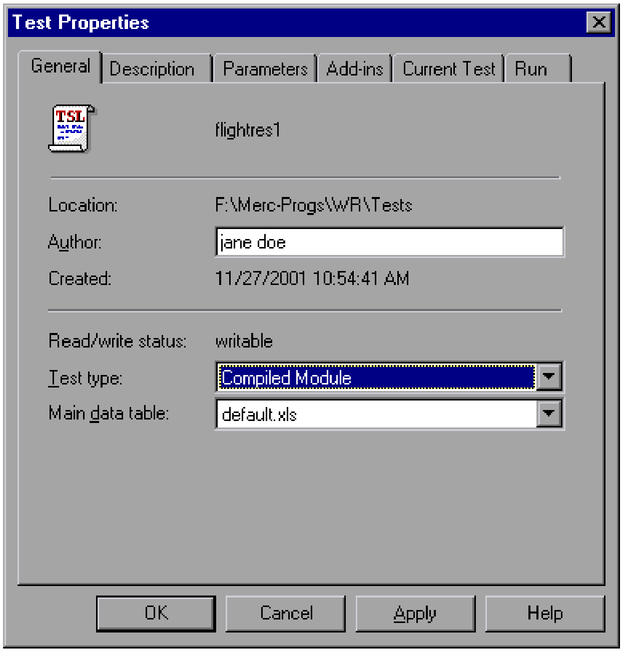
For more information, see "Example of a Compiled Module" on page 174.

Creating a Compiled Module

Creating a compiled module is similar to creating a regular test script.

To create a compiled module:

1. Choose **File** > **Open** to open a new test.
2. Write the user-defined functions in the test.
3. Choose **File** > **Test Properties** and click the **General** tab.
4. In the **Test type** list, choose **Compiled Module** and then click **OK**.



flightres1

Location: F:\Merc-Progs\WR\Tests

Author: jane doe

Created: 11/27/2001 10:54:41 AM

Read/write status: writable

Test type: Compiled Module

Main data table: default.xls

5 Choose **File** > **Save**.

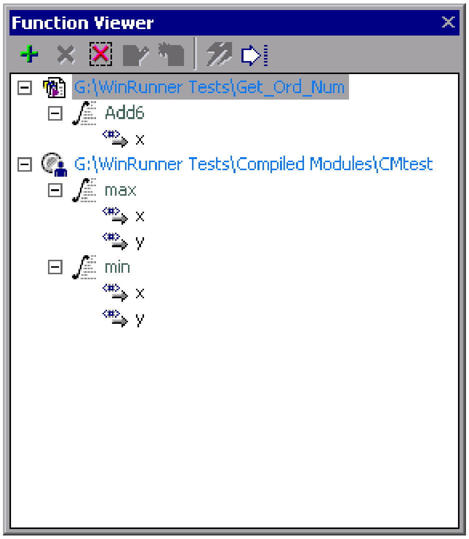
Save your module in a location that is readily available to all tests that may call functions from it. When a module is loaded, WinRunner locates it according to the search path you define. For more information on defining a search path, see "Setting the Search Path" on page 136.

6 Compile the module using the **load** function in a test script, or the **Load** button in the Function Viewer. For more information, see "Loading and Unloading a Compiled Module" on page 170.

Using the Function Viewer

You can use the Function Viewer to load and unload compiled modules, to copy, paste and execute the functions of loaded compiled modules and tests, and to open loaded compiled modules and tests containing loaded functions.

The Function Viewer is a dockable window that can be opened or closed at any time.


Function Viewer
G:\WinRunner Tests\Get_Ord_Num
Add6
x
G:\WinRunner Tests\Compiled Modules\CMtest
max
x
y
min
x
y

The Function Viewer is comprised of a toolbar and a pane that displays the function tree.

The function tree has three levels. At the highest level you can see the loaded compiled modules and open tests containing loaded functions.

A test, or a compiled module that was loaded using the **Run** toolbar button, is indicated by the test icon.

A compiled module that was loaded by the **load** function in a test or by the Function Viewer **Load** button is indicated by the compiled module icon.

A non-static function is indicated by the non-static function icon, one level below the compiled modules and the tests in the function tree.

A static function is indicated by the static function icon, one level below the compiled modules and the tests in the function tree.

Input parameters, if any, are indicated by the input parameter icon, one level below the displayed functions in the viewer.

Output and inout parameters, if any, are indicated by the output parameter icon, one level below the displayed functions in the viewer.

You can expand any item by clicking the **Expand** button beside it or by pressing the plus (+) key on your keyboard number pad.

You can expand any item and all levels below it by pressing the asterisk (*) key on your keyboard number pad.

You can collapse any item and all levels below it by clicking the **Collapse** button beside it or by pressing the minus (-) key on your keyboard number pad.

The Function Viewer toolbar provides the following options:

- **Load**—Enables you to load a compiled module. For more information, see “Loading a Function or Compiled Module” on page 163.
- **Unload**—Unloads the currently selected compiled module.
- **Unload All Modules**—Unloads all compiled modules. This button does not have any effect on functions loaded from tests.
- **Copy**—Copies the selected function prototype to the clipboard. For more information, see “Copying and Pasting a Function Prototype” on page 165.

- **Paste**—Copies and pastes the selected function prototype to the current cursor location in the test. For more information, see “Copying and Pasting a Function Prototype” on page 165.
- **Execute**—Executes the selected function. For more information, see “Executing a Function from the Function Viewer” on page 165.
- **Go To**—Opens the selected compiled module, test, or function in the test window. For more information, see “Opening a Loaded Compiled Module, Test, or Function for Viewing or Editing” on page 166.

Displaying the Function Viewer

You can display the Function Viewer by choosing **Tools** > **Function Viewer**. You can dock the Function Viewer to the top, bottom or either side of the test window. Close the Function Viewer by clicking the **Close** toolbar button, or by choosing **Tools** > **Function Viewer** again.

Loading a Function or Compiled Module

If you want to call a function that is not defined in the calling test, you need to load it.

You can define functions in tests, or in compiled modules.

You load a function defined in a test by running the test. When you run a test, all functions that are defined in that test are loaded and continue to be available until you click the **Stop** button. The functions continue to be loaded even if the test is closed. When you click the **Stop** button, you unload all functions loaded from all tests that are not compiled modules. For more information, see “Employing Functions Defined In Tests” on page 167.

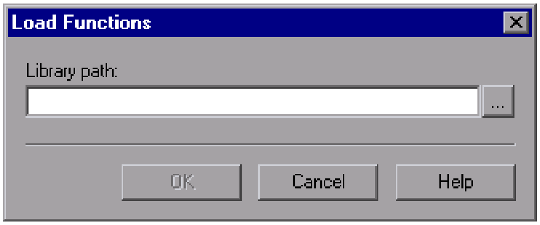
Note: If you load a compiled module that contains a function that has the same name as a function in an already-loaded compiled module, the function from the first compiled module is unloaded, and the function from the second compiled module is loaded. If you load a compiled module that contains a function that has the same name as a standard TSL function, the original function is overridden.

You load all the functions defined in a compiled module in one of two ways:

- From a test script. You can load a compiled module using the **load** TSL function in a test script. For more information, see "Loading and Unloading a Compiled Module Using the TSL Functions" on page 171.
- From the Function Viewer. You can load a compiled module using the **Load** toolbar button in the Function Viewer. Once the compiled module is loaded, you can call any of the non-static functions defined in it.

To load a compiled module from the Function Viewer:

1. Make sure that the Function Viewer is visible. See "Displaying the Function Viewer" on page 163.
2. Click the **Load** button. The Load Functions dialog box opens.



3. Use the browse button to find the compiled module you want to load or enter the path manually in the **Library path** edit box.
4. Click **OK**. The compiled module is displayed in the Function Viewer tree. All its functions are loaded.

Unloading a Function or Compiled Module

To unload a compiled module, select it and click the **Unload** button.

To unload all the loaded compiled modules, click the **Unload All Modules** button.

The **Unload** and **Unload All** buttons cannot be used to unload the functions in a test. To unload these functions, click the **Stop** button.

Copying and Pasting a Function Prototype

You can copy a function prototype to the clipboard and then paste it to any other application.

To copy a function prototype to the clipboard, simply select it and click the **Copy** button.

To copy and paste a function prototype to the test screen, select it, place the cursor at the selected position on the test screen, then click the **Paste** button. You do not need to copy it first. You can also drag-and-drop the function prototype.

Executing a Function from the Function Viewer

You can execute a non-static function directly from the Function Viewer. This is useful for testing your functions. For example, if you are creating a compiled module with many functions and you want to test just one function, you can execute the function directly without having to write a test to load it and call it.

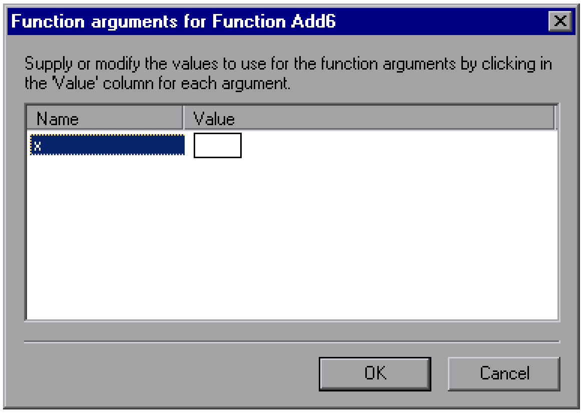
Note: A static function cannot be executed from the Function Viewer. If you try to execute a static function, an error message is displayed

---

To execute a function from the Function Viewer

1. Select the function you want to execute, in the Function Viewer.
2. Click the **Execute** toolbar button. If the function does not require input parameters, the function runs.

3 If the function requires parameters, the Function Arguments dialog box opens.



Function arguments for Function Add6

Supply or modify the values to use for the function arguments by clicking in the 'Value' column for each argument.

Name	Value
x	

4 Enter values for the parameters by clicking in the row of each argument under the **Value** column and then typing the value. Click **OK**. The function runs.

Note: When you call a function containing an output or an inout parameter, the argument corresponding to that parameter must be a variable, and not an expression.

When you call a function containing an input parameter, the argument corresponding to that parameter cannot be a variable, but must be a string or a number. Any non-numeric characters will be treated as a string.

Opening a Loaded Compiled Module, Test, or Function for Viewing or Editing

You can use the **Go To** toolbar button to open any loaded compiled module, test or function that is displayed in the Function Viewer. You can then view and edit the content.

To open a compiled module or test in the test window

1 Select the compiled module, test, or function in the Function Viewer.

2 Click the **Go To** button or double-click the compiled module, test, or function. The compiled module or test is opened in its own tab in the test window. If you open a function, the entire test in which the function is defined is opened, and the function's first line is marked by the execution marker.

Employing Functions Defined In Tests

You can define functions in any test script. As you run the test, WinRunner loads each function defined in the test. All functions defined in the test are loaded, even those that are not called by the test. The loaded functions are displayed in the Function Viewer.

Note: If an error prevents WinRunner from reading a function when you run the test, then that function is not loaded and is not displayed in the Function Viewer.

The functions loaded from a test are available from the time the test runs until the end of the WinRunner session, or until you click the **Stop** button.

■ Stop

When you click the **Stop** button at any time, all tests and their functions no longer appear in the Function Viewer and their functions cannot be called by other tests.

While a function is displayed in the function viewer, you can call the function, execute it, copy and paste the function prototype or open the function. For more information on copying and pasting function prototypes, see "Copying and Pasting a Function Prototype" on page 165. For more information on executing loaded functions, see "Executing a Function from the Function Viewer" on page 165. For more information on opening a function, see "Opening a Loaded Compiled Module, Test, or Function for Viewing or Editing" on page 166.

Employing Functions Defined in Compiled Modules

A compiled module is a script containing a library of user-defined functions that you want to call frequently from other tests.

By saving functions in compiled modules, you make it easier for other tests to call those functions.

When you load a compiled module from a test, you can load it as a system module, or as a user module. System modules are invisible to the tester, and contain finished, working, frequently-used functions. User modules are still in development or have less common uses.

You can load compiled modules from your startup test.

Understanding Compiled Modules

When you load a compiled module, its non-static functions are automatically compiled and remain in memory. You can call them directly from within any test.

For instance, you can create a compiled module containing functions that compare the size of two files, or check your system's current memory resources.

Compiled modules can improve the organization and performance of your tests. Since you debug compiled modules before using them, tests that call functions from these modules require less error-checking. In addition, calling a function that is already compiled is significantly faster than interpreting a function in a test script.

Note: If you are working in the *GUI Map File per Test* mode, compiled modules do not load GUI map files. If your compiled module references GUI objects, then those objects must also be referenced in the test that loads the compiled module. For more information, refer to Chapter 6, "Working in the GUI Map File per Test Mode" in the *Mercury WinRunner Basic Features User's Guide*.

Understanding System and User Compiled Modules

A compiled module can be loaded as a system compiled module or a user compiled module.

- A *system compiled module* is a closed module that is not visible to the tester. It is not displayed in the test window when it is loaded, cannot be stepped into, and cannot be stopped by a pause command. A system module is not unloaded when you execute an **unload** statement with no parameters (global unload).

  You can select whether to display loaded system modules in the Function Viewer. By default they are not displayed. To display the system modules choose **Tools** > **General Options** and select the **Display System Modules** box. The option takes effect the next time you open WinRunner.

- A user compiled module is the opposite of a system module in most respects. It is displayed when it runs and you can use all WinRunner debugging options to control the run. Generally, a user module is one that is still being developed. In such a module you might want to make changes and compile them incrementally.

You define compiled modules as either system or user when you load them using the **load** TSL function in a test script. For more information, see "Loading and Unloading a Compiled Module Using the TSL Functions" on page 171. When you load a compiled module using the **Load** button in the Function Viewer, it is always loaded as a user compiled module.

Running Compiled Modules Automatically on WinRunner Startup

If you create a compiled module that contains frequently-used functions, you can load it from your startup test. You do this by adding **load** statements to your startup test. For more information, see Chapter 23, "Initializing Special Configurations."

You do not need to add **load** statements to your startup test or to any other test to load the *recovery compiled module*. The recovery compiled module is automatically loaded when you start WinRunner. For more information on the recovery compiled module, see Chapter 4, "Defining and Using Recovery Scenarios."

Loading and Unloading a Compiled Module

To access the functions in a compiled module you need to load the module. You can load a module in one of three ways:

- Load the module using the **Load** button in the Function Viewer.
- Load the module from a test script using the TSL **load** or **reload** functions. Any test script can load a compiled module using the **load** or **reload** functions.
- Run the module script using the WinRunner Run commands.

When you run a compiled module, it is loaded into memory with all its functions, and can be seen in the Function Viewer. To unload a module loaded this way, click the **Stop** button. The **Unload** and **Unload All Modules** buttons do not work for a module that was loaded using the **Run** button.

If you need to debug a module or make changes, you can use the Step command to perform incremental compilation. You only need to run the part of the module that was changed in order to update the entire module.

The remainder of this section addresses the first two options above.

Loading and Unloading a Compiled Module Using the Function Viewer

To load or unload a compiled module using the Function Viewer, use the **Load** or **Unload** toolbar buttons. This is useful especially for debugging individual tests that are usually part of a larger call chain. For example, suppose the first test in a call chain loads all of the compiled modules for all of the called tests in a chain. If you want to debug one test in the chain, you can load compiled modules using the Function Viewer instead of running another test to load the modules.

When you load a compiled module using the **Load** button, it is loaded as a user compiled module.

When you unload a compiled module using the **Unload** or **Unload All** buttons from the Function Viewer, a single click on the toolbar button completely clears the compiled module(s) from the memory. To unload only individual instances of a loaded module, use the **unload** TSL function. For more information, see “Loading and Unloading a Compiled Module Using the TSL Functions” on page 171.

Loading and Unloading a Compiled Module Using the TSL Functions

You can load a compiled module from within any test script using the **load** command; all tests will then be able to access the function until you exit WinRunner or unload the compiled module.

You should insert **load** commands into tests so that you can run them unsupervised. For example, suppose you have finished debugging a test. While debugging the test, you used the Function Viewer to load and unload any modules you needed. To run the test unsupervised, you must now add **load** statements to load the necessary modules programmatically, either in the test that calls the functions, or on a previous test in a call chain.

If you try to load a module that has already been loaded using the TSL function, WinRunner does not load it again. Instead, it initializes variables and increments a *load counter*. If a module has been loaded more than once, then the **unload** statement does not unload the module, but rather decrements the counter.

For example, suppose that Test A loads the module *math_functions*, and then calls Test B. Test B also loads *math_functions*, and then unloads it at the end of the test. After Test B runs, Test A calls functions defined in *math_functions*. Suppose also that the unload function, instead of decrementing the counter, were to completely unload the compiled module. In such a case, Test B's **unload** function would completely unload *math_functions* from memory, and then the subsequent calls to *math_functions* by Test A would fail.

The counter exists to avoid this situation. With the counter, when Test B unloads *math_functions*, it decrements the counter, but *math_functions* is still resident in memory for any subsequent calls from Test A.

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

load (*module_name* [,*module_type*] [,*open_status*]);

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 function module is a system module or a user module: 1 indicates a system module; 0 indicates a user module.

(Default = 0)

For more information on system and user modules, see “Understanding System and User Compiled Modules” on page 169.

The second optional parameter indicates whether a *user* module will remain open in the WinRunner window or will close automatically after it is loaded: 1 indicates that the module will close automatically; 0 indicates that the module will remain open.

(Default = 0)

When the **load** function is executed for the first time, the module is compiled and stored in memory. This module is ready for use by any test and does not need to be reinterpreted.

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

➤ The **unload** function removes the latest instance of a loaded module or selected functions from memory. It has the following syntax:

unload ([*module_name* **|** *test_name* [, "*function_name*"]] **);**

For example, the following statement removes all functions loaded within the compiled module named mem_test.

```
unload ("mem_test");
```

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

If a module was loaded more than once by different scripts, then a separate unload statement is required for each load. For more information, see “Loading and Unloading a Compiled Module Using the TSL Functions” on page 171.

➤ If you make changes in a module, you should 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* [,*module_type*] [,*open_status*] **);**

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 a *user* module will remain open in the WinRunner window or will close automatically after it is loaded. 1 indicates that the module will close automatically. 0 indicates that the module will remain open.

(Default = 0)

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

Example of a Compiled Module

The following module contains two simple, all-purpose functions that you can call from any test. The first function receives a pair of numbers and returns the number with the higher value. The second function receives a pair of numbers and returns the one with the lower 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 y;
    else
        return x;
}
```

12

Calling Functions from External Libraries

WinRunner enables you to call functions from the Windows API and from any external DLL (Dynamic Link Library).

This chapter describes:

- About Calling Functions from External Libraries
- Dynamically Loading External Libraries
- Declaring External Functions in TSL
- Windows API Examples

About Calling Functions from External Libraries

You can extend the power of your automated tests by calling functions from the Windows API or from any external DLL. For example, using functions in the Windows API you can:

- Use a standard Windows message box in a test with the *MessageBox* function.
- Send a WM (Windows Message) message to the application being tested with the *SendMessage* function.
- Retrieve information about your application's windows with the *GetWindow* function.
- Integrate the system beep into tests with the *MessageBeep* function.
- Run any windows program using *ShellExecute*, and define additional parameters such as the working directory and the window size.

- Check the text color in a field in the application being tested with *GetTextColor*. This can be important when the text color indicates operation status.
- Access the Windows clipboard using the *GetClipboard* functions.

You can call any function exported from a DLL with the _ _ stdcall calling convention. You can also load DLLs that are part of the application being tested in order to access its exported functions.

Using the **load_dll** function, you dynamically load the libraries containing the functions you need. Before you actually call the function, you must write an *extern* declaration so that the interpreter knows that the function resides in an external library.

Note: For information about specific Windows API functions, refer to the *Windows API Reference*. For examples of using the Windows API functions in WinRunner test scripts, refer to the *read.me* file in the *\lib\win32api* folder in the installation folder.

Dynamically Loading External Libraries

In order to load the external DLLs (Dynamic Link Libraries) containing the functions you want to call, use the TSL function **load_dll**. This function performs a runtime load of a 32-bit DLL. It has the following syntax:

load_dll (*pathname* **);**

The *pathname* is the full pathname of the DLL to be loaded.

For example:

```
load_dll ("h:\\qa_libs\\os_check.dll");
```

To unload a loaded external DLL, use the TSL function **unload_dll**. It has the following syntax:

unload_dll (*pathname* **);**

For example:

```
unload_dll ("h:\\qa_libs\\os_check.dll");
```

The *pathname* is the full pathname of the 32-bit DLL to be unloaded.

To unload all loaded 32-bit DLLs from memory, use the following statement:

```
unload_dll ("");
```

For more information, refer to the *TSL Reference*.

Declaring External Functions in TSL

You must write an *extern* declaration for each function you want to call from an external library. The extern declaration must appear before the function call. It is recommended to store these declarations in a startup test. (For more information on startup tests, see Chapter 23, "Initializing Special Configurations.")

The syntax of the extern declaration is:

extern *type function_name* **(** *parameter1*, *parameter2*,... **);**

The *type* refers to the return value of the function. The type can be one of the following:

char (signed and unsigned)	*float*
short (signed and unsigned)	*double*
int (signed and unsigned)	*string* (equivalent to C char*)

Each *parameter* must include the following information:

[mode]   *type*   [name]   [<*size*>]

The *mode* can be either *in*, *out*, or *inout*. The default is *in*. Note that these values must appear in lowercase letters.

The *type* can be any of the values listed above.

An optional *name* can be assigned to the parameter to improve readability.

The <*size*> is required only for an *out* or *inout* parameter of type *string* (see below).

For example, suppose you want to call a function called set_clock that sets the time on a clock application. The function is part of an external DLL that you loaded with the **load_dll** statement. To declare the function, write:

```
extern int set_clock (string name, int time);
```

The set_clock function accepts two parameters. Since they are both input parameters, no mode is specified. The first parameter, a string, is the name of the clock window. The second parameter specifies the time to be set on the clock. The function returns an integer that indicates whether the operation succeeded.

Once the extern declaration is interpreted, you can call the set_clock function the same way you call a TSL function:

```
result = set_clock ("clock v. 3.0", 3);
```

If an extern declaration includes an *out* or *inout* parameter of type *string*, you must budget the maximum possible string size by specifying an integer <*size*> after the parameter *type* or (optional) *name*. For example, the statement below declares the function get_clock_string, that returns the time displayed in a clock application as a string value in the format “The time is...”.

```
extern int get_clock_string (string clock, out string time <20>);
```

The *size* should be large enough to avoid an overflow. If no value is specified for *size*, the default is 100.

TSL identifies the function in your code by its name only. You must pass the correct parameter information from TSL to the function. TSL does not check parameters. If the information is incorrect, the operation fails.

Note: If you want to return a string value from a function in an external DLL, it is recommended to use an output parameter rather than a return value.

For example your DLL should look something like:

```
int foo(char* szRetString)
{
  ...
  strcpy(szRetString, "hi");
  return nErrCode;
}
```

And the corresponding **extern** statement should be something like:

```
extern int foo(out string);
```

In addition, your external function must adhere to the following conventions:

- Any parameter designated as a *string* in TSL must correspond to a parameter of type *char**.
- Any parameter of mode *out* or *inout* in TSL must correspond to a pointer in your exported function. For instance, a parameter *out int* in TSL must correspond to a parameter *int** in the exported function.
- The external function must observe the standard Pascal calling convention *export far Pascal*.

For example, the following declaration in TSL:

```
extern int set_clock (string name, inout int time);
```

must appear as follows in your external function:

```
int set_clock(
        char* name,
        int* time
        );
```

Windows API Examples

The following sample tests call functions in the Windows API.

Checking Window Mnemonics

This test integrates the API function *GetWindowTextA* into a TSL function that checks for mnemonics (underlined letters used for keyboard shortcuts) in object labels. The TSL function receives one parameter: the logical name of an object. If a mnemonic is not found in an object's label, a message is sent to a report.

```
# load the appropriate DLL (from Windows folder)
load ("win32api");

# define the user-defined function "check_labels"
public function check_labels(in obj)
{
    auto hWnd,title,pos,win;
    win = GUI_get_window();
    obj_get_info(obj,"handle",hWnd);
    GetWindowTextA(hWnd,title,128);
     pos = index(title,"&");
    if (pos == 0)
        report_msg("No mnemonic for object: "& obj & "in window: "& win);
}
```

```
# start Notepad application
invoke_application("notepad.exe","","",SW_SHOW);

# open Find window
set_window ("Notepad");
menu_select_item ("Search;Find...");

# check mnemonics in "Up" radio button and "Cancel" pushbutton
set_window ("Find");
check_labels ("Up");
check_labels ("Cancel");
```

Loading a DLL and External Function

This test fragment uses crk_w.dll to prevent recording on a debugging application. To do so, it calls the external *set_debugger_pid* function.

```
# load the appropriate DLL
load_dll("crk_w.dll");

# declare function
extern int set_debugger_pid(long);

# load Systems DLLs (from Windows folder)
load ("win32api");

# find debugger process ID
win_get_info("Debugger","handle",hwnd);
GetWindowThreadProcessId(hwnd,Proc);

# notify WinRunner of the debugger process ID
set_debugger_pid(Proc);
```

13

Creating Dialog Boxes for Interactive Input

WinRunner enables you to create dialog boxes that you can use to pass input to your test during an interactive test run.

This chapter describes:

About Creating Dialog Boxes for Interactive Input

You can create dialog boxes that pop up during an interactive test run, prompting the user to perform an action—such as typing in text or selecting an item from a list. This is useful when the user must make a decision based on the behavior of the application under test (AUT) during runtime, and then enter input accordingly. For example, you can instruct WinRunner to execute a particular group of tests according to the user name that is typed into the dialog box.

To create the dialog box, you enter a TSL statement in the appropriate location in your test script. During an interactive test run, the dialog box opens when the statement is executed. By using control flow statements, you can determine how WinRunner responds to the user input in each case.

There are five different types of dialog boxes that you can create using the following TSL functions:

- **create_input_dialog** creates a dialog box with any message you specify, and an edit field. The function returns a string containing whatever you type into the edit field, during an interactive run.
- **create_list_dialog** creates a dialog box with a list of items, and your message. The function returns a string containing the item that you select during an interactive run.
- **create_custom_dialog** creates a dialog box with edit fields, check boxes, an "execute" button, and a Cancel button. When the "execute" button is clicked, the **create_custom_dialog** function executes a specified function.
- **create_browse_file_dialog** displays a browse dialog box from which the user selects a file. During an interactive run, the function returns a string containing the name of the selected file.
- **create_password_dialog** creates a dialog box with two edit fields, one for login name input, and one for password input. You use a password dialog box to limit user access to tests or parts of tests.

Each dialog box opens when the statement that creates it is executed during a test run, and closes when one of the buttons inside it is clicked.

Creating an Input Dialog Box

An input dialog box contains a custom one-line message, an edit field, and OK and Cancel buttons. The text that the user types into the edit field during a test run is returned as a string.

You use the TSL function **create_input_dialog** to create an input dialog box. This function has the following syntax:

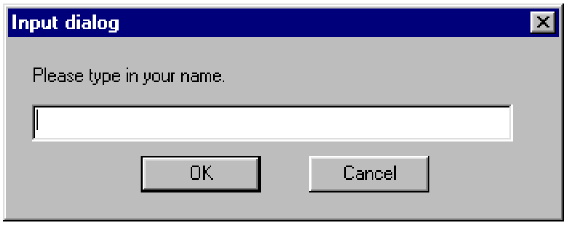
create_input_dialog (*message* **);**

The *message* can be any expression. The text appears as a single line in the dialog box.

For example, you could create an input dialog box that asks for a user name. This name is returned to a variable and is used in an **if** statement in order to call a specific test suite for any of several users.

To create such a dialog box, you would program the following statement:

```
name = create_input_dialog ("Please type in your name.");
```



The input that is typed into the dialog box during a test run is passed to the variable *name* when the **OK** button is clicked. If the **Cancel** button is clicked, an empty string (empty quotation marks) is passed to the variable *name*.

Tip: You can use the following statements to display the message that the user types in the dialog box:

```
rc=create_input_dialog("Message");
pause(rc);
```

For additional information on the **pause** function, refer to the *TSL Reference*.

Note that you can precede the message parameter with an exclamation mark. When the user types input into the edit field, each character entered is represented by an asterisk. Use an exclamation mark to prevent others from seeing confidential information.

Creating a List Dialog Box

A list dialog box has a title and a list of items that can be selected. The item selected by the user from the list is passed as a string to a variable.

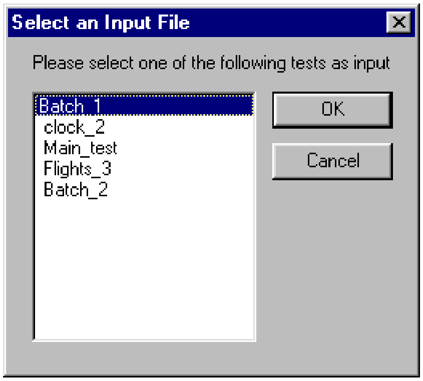
You use the TSL function **create_list_dialog** to create a list dialog box. This function has the following syntax:

create_list_dialog (*title*, *message*, *list_items* **);**

- *title* is an expression that appears in the window banner of the dialog box.
- *message* is one line of text that appear in the dialog box.
- *list_items* contains the options that appear in the dialog box. Items are separated by commas, and the entire list is considered a single string.

For example, you can create a dialog box that allows the user to select a test to open. To do so, you could enter the following statement:

```
filename = create_list_dialog ("Select an Input File", "Please select one of the following tests as input", "Batch_1, clock_2, Main_test, Flights_3, Batch_2");
```


Select an Input File

Please select one of the following tests as input

Batch_1
clock_2
Main_test
Flights_3
Batch_2

The item that is selected from the list during a test run is passed to the variable *filename* when the **OK** button is clicked. If the **Cancel** button is clicked, an empty string (empty quotation marks) is passed to the variable *filename*.

Creating a Custom Dialog Box

A custom dialog box has a custom title, up to ten edit fields, up to ten check boxes, an "execute" button, and a Cancel button. You specify the label for the "execute" button. When you click the "execute" button, a specified function is executed. The function can be either a TSL function or a user-defined function.

You use the TSL function **create_custom_dialog** to create a custom dialog box. This function has the following syntax:

create_custom_dialog (*function_name*, *title*, *button_name*, *edit_name*$_{1\text{-}n}$, *check_name*$_{1\text{-}m}$ **);**

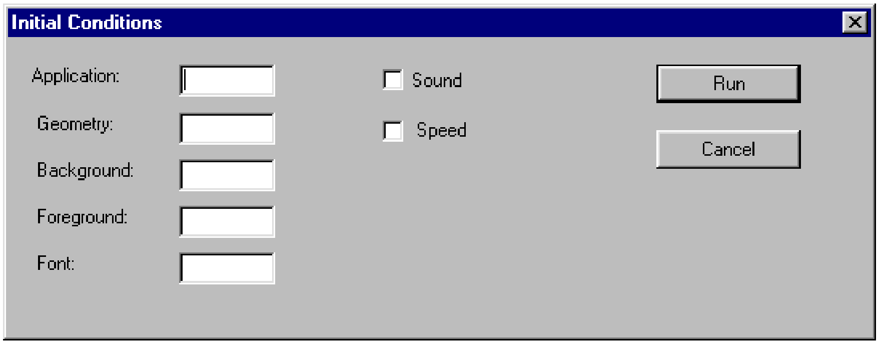
- ➤ *function_name* is the name of the function that is executed when you click the "execute" button.
- ➤ *title* is an expression that appears in the title bar of the dialog box.
- ➤ *button_name* is the label that will appear on the "execute" button. You click this button to execute the contained function.
- ➤ *edit_name* contains the labels of the edit field(s) of the dialog box. Multiple edit field labels are separated by commas, and all the labels together are considered a single string. If the dialog box has no edit fields, this parameter must be an empty string (empty quotation marks).
- ➤ *check_name* contains the labels of the check boxes in the dialog box. Multiple check box labels are separated by commas, and all the labels together are considered a single string. If the dialog box has no check boxes, this parameter must be an empty string (empty quotation marks).

When the "execute" button is clicked, the values that the user enters are passed as parameters to the specified function, in the following order:

edit_name$_1$*,... edit_name*$_n$ *,check_name*$_1$*,... check_name*$_m$

In the following example, the custom dialog box allows the user to specify startup parameters for an application. When the user clicks the **Run** button, the user-defined function, run_application1, invokes the specified Windows application with the initial conditions that the user supplied.

```
res = create_custom_dialog ("run_application1", "Initial Conditions", "Run",
    "Application:, Geometry:, Background:, Foreground:, Font:", "Sound,
    Speed");
```



If the specified function returns a value, this value is passed to the variable *res*. If the **Cancel** button is clicked, an empty string (empty quotation marks) is passed to the variable *res*.

Note that you can precede any edit field label with an exclamation mark. When the user types input into the edit field, each character entered is represented by an asterisk. You use an exclamation mark to prevent others from seeing confidential information, such as a password.

Creating a Browse Dialog Box

A browse dialog box allows you to select a file from a list of files, and returns the name of the selected file as a string.

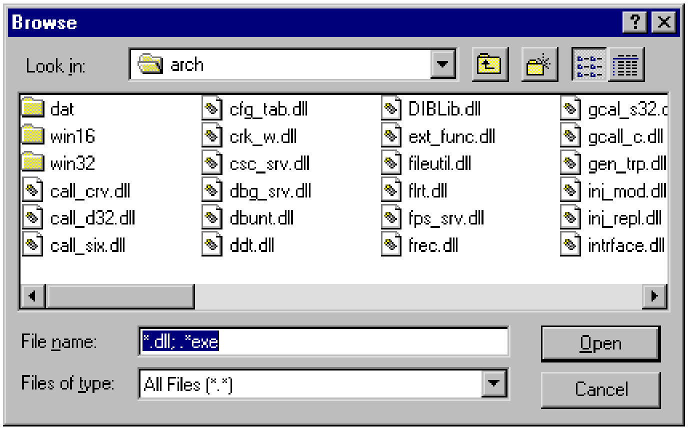
You use the TSL function **create_browse_file_dialog** to create a browse dialog box. This function has the following syntax:

create_browse_file_dialog (*filter* **);**

where *filter* sets a filter for the files to display in the Browse dialog box. You can use wildcards to display all files (*.*) or only selected files (*.exe or *.txt etc.).

In the following example, the browse dialog box displays all files with extensions .dll or .exe.

```
filename = create_browse_file_dialog( "*.dll;*.exe" );
```



Look in: arch

File name: *.dll; .*exe

Files of type: All Files (*.*)

When the **Open** button is clicked, the name and path of the selected file is passed to the variable *filename*. If the **Cancel** button is clicked, an empty string (empty quotation marks) is passed to the variable *filename*.

Creating a Password Dialog Box

A password dialog box has two edit fields, an OK button, and a Cancel button. You supply the labels for the edit fields. The text that the user types into the edit fields during the interactive test run is saved to variables for analysis.

You use the TSL function **create_password_dialog** to create a password dialog box. This function has the following syntax:

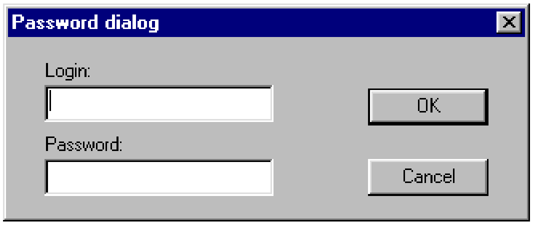
create_password_dialog (*login, password, login_out, password_out* **);**

➤ *login* is the label of the first edit field, used for user-name input. If you specify an empty string (empty quotation marks), the default label “Login” is displayed.

- *password* is the label of the second edit field, used for password input. If you specify an empty string (empty quotation marks), the default label "Password" is displayed. When the user enters input into this edit field, the characters do not appear on the screen, but are represented by asterisks.
- *login_out* is the name of the parameter to which the contents of the first edit field (login) are passed. Use this parameter to verify the contents of the login edit field.
- *password_out* is the name of the parameter to which the contents of the second edit field (password) are passed. Use this parameter to verify the contents of the password edit field.

The following example shows a password dialog box created using the default edit field labels.

```
status = create_password_dialog ("", "", user_name, password);
```



If the **OK** button is clicked, the value 1 is passed to the variable *status*. If the **Cancel** button is clicked, the value 0 is passed to the variable *status* and the *login_out* and *password_out* parameters are assigned empty strings.

Part IV

Running Tests—Advanced

14

Running Batch Tests

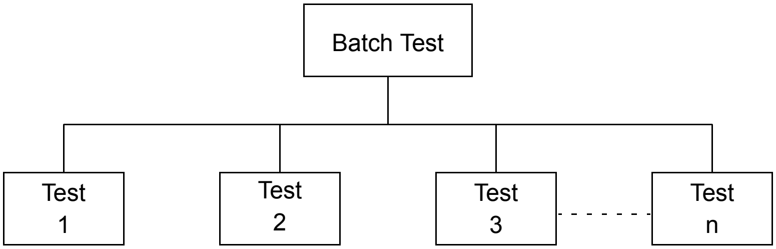
WinRunner enables you to execute a group of tests unattended. This can be particularly useful when you want to run a large group of tests overnight or at other off-peak hours.

This chapter describes:

- About Running Batch Tests
- Creating a Batch Test
- Running a Batch Test
- Storing Batch Test Results
- Viewing Batch Test Results

About Running Batch Tests

You can run a group of tests unattended by creating and executing a single batch test. A batch test is a test script that contains call statements to other tests. It opens and executes each test and saves the test results.



A batch test looks like a regular test that includes call statements. A test becomes a "batch test" when you select the **Run in batch mode** option in the **Run** category of the General Options dialog box before you execute the test.

When you run a test using the **Verify** run option in Batch mode, WinRunner suppresses all messages that would ordinarily be displayed during the test run, such as a message reporting a bitmap mismatch. WinRunner also suppresses all **pause** statements and any halts in the test run resulting from run time errors.

By suppressing all messages, WinRunner can run a batch test unattended. This differs from a regular, interactive test run in which messages appear on the screen and prompt you to click a button in order to resume test execution. A batch test enables you to run tests overnight or during off-peak hours, so that you can save time while testing your application.

Note: Messages are suppressed for a batch test only if you run the test using the **Verify** run mode. If you use the **Update** or **Debug** run mode to run the test, messages are displayed even when the **Run in batch mode** option is selected.

At each break during a test run—such as after a Step command, at a breakpoint, or at the end of a test, you can view the current chain of called tests in the Call Chain pane of the Debug Viewer window. For more information, see "Viewing the Call Chain" on page 145

When a batch test run is completed, you can view the results in the Test Results window. The window displays the results of all the major events that occurred during the run.

Note that you can also run a group of tests from the command line. For details, see Chapter 15, "Running Tests from the Command Line."

Creating a Batch Test

A batch test is a test script that calls other tests. You program a batch test by typing call statements directly into the test window and selecting the **Run in batch mode** option in the **Run** category of the General Options dialog box before you execute the test.

A batch test may include programming elements such as loops and decision-making statements. Loops enable a batch test to run called tests a specified number of times. Decision-making statements such as *if/else* and *switch* condition test execution on the results of a test called previously by the same batch script. See Chapter 7, "Enhancing Your Test Scripts with Programming," for more information.

For example, the following batch test executes three tests in succession, then loops back and calls the tests again. The loop specifies that the batch test should call the tests ten times.

```
for (i=0; i<10; i++)
    {
    call "c:\\pbtests\\open" ();
    call "c:\\pbtests\\setup" ();
    call "c:\\pbtests\\save" ();
    }
```

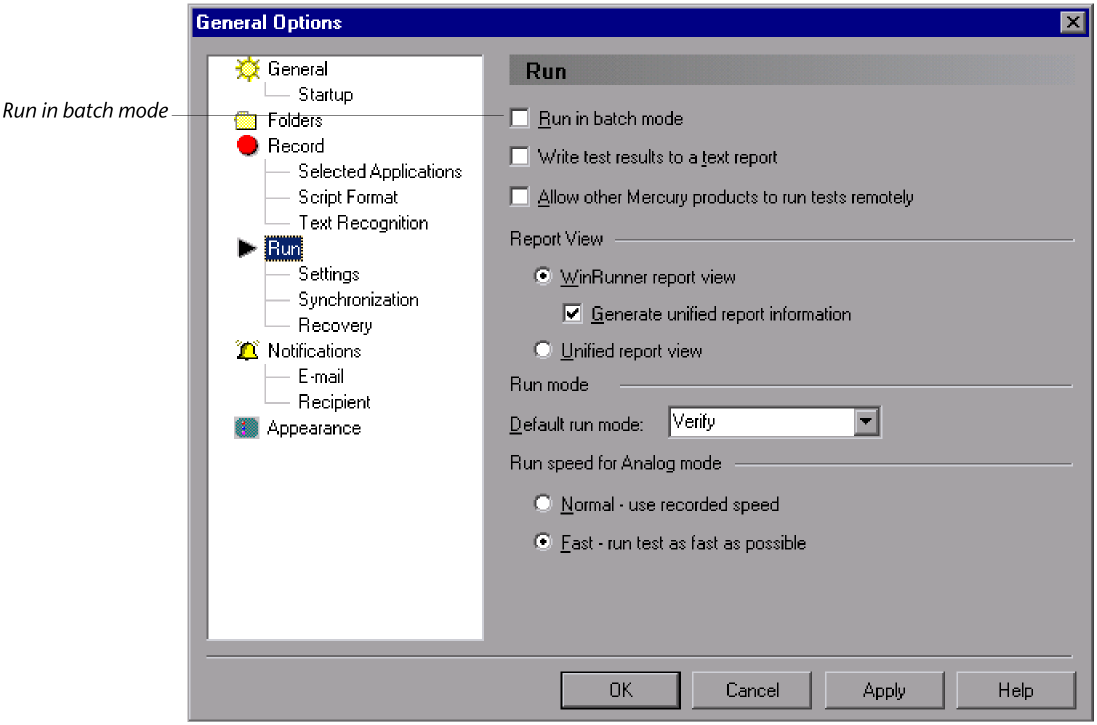
To enable a batch test:

1 Choose **Tools** > **General Options**.

The General Options dialog box opens.

2 Click the **Run** category.

3 Select the **Run in batch mode** check box.



Run in batch mode

General Options

- General
  - Startup
- Folders
- Record
  - Selected Applications
  - Script Format
  - Text Recognition
- Run
  - Settings
  - Synchronization
  - Recovery
- Notifications
  - E-mail
  - Recipient
- Appearance

Run

□ Run in batch mode

□ Write test results to a text report

□ Allow other Mercury products to run tests remotely

Report View

● WinRunner report view

☑ Generate unified report information

○ Unified report view

Run mode

Default run mode: Verify

Run speed for Analog mode

○ Normal - use recorded speed

● Fast - run test as fast as possible

4 Click **OK** to close the General Options dialog box.

For more information on setting the batch option in the General Options dialog box, refer to Chapter 23, “Setting Global Testing Options” in the *Mercury WinRunner Basic Features User’s Guide*.

Running a Batch Test

You execute a batch test in the same way that you execute a regular test. Choose a run mode (**Verify**, **Update**, or **Debug**) from the list on the toolbar and choose **Test** > **Run from Top**. Refer to Chapter 20, "Understanding Test Runs" in the *Mercury WinRunner Basic Features User's Guide* for more information.

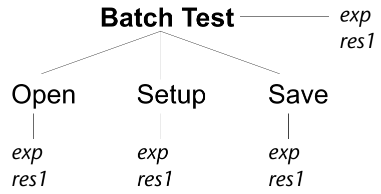
When you run a batch test, WinRunner opens and executes each called test. All messages are suppressed so that the tests are run without interruption. If you run the batch test in **Verify** mode, the current test results are compared to the expected test results saved earlier. If you are running the batch test in order to update expected results, new expected results are created in the expected results folder for each test. See "Storing Batch Test Results" below for more information. When the batch test run is completed, you can view the test results in the Test Results window.

Note that if your tests contain TSL **texit** statements, WinRunner interprets these statements differently for a batch test run than for a regular test run. During a regular test run, **texit** terminates test execution. During a batch test run, **texit** halts execution of the current test only and control is returned to the batch test.

Storing Batch Test Results

When you run a regular, interactive test, results are stored in a subfolder under the test. If **Run in batch mode** is selected in the **Run** category of the General Options dialog box, then WinRunner saves the results for each (top-level) called test separately in a subfolder under the called test. Additionally, a subfolder is also created for the batch test that contains the results of the entire batch test run, including all called tests.

For example, suppose you create three tests: *Open*, *Setup*, and *Save*. For each test, expected results are saved in an *exp* subfolder under the test folder. Suppose you also create a batch test that calls the three tests. Before running the batch test in **Verify** mode, you instruct WinRunner to save the results in a subfolder of the calling test called *res1*. When the batch test is run, it compares the current test results to the expected results saved earlier. Under each test folder, WinRunner creates a subfolder called *res1* in which it saves the verification results for the test. A *res1* folder is also created under the batch test to contain the overall verification results for the entire run.



If you run the batch test in **Update** mode in order to update expected results, WinRunner overwrites the expected results in the *exp* subfolder for each test and for the batch test.

Notes:

If a called test already had a folder called *res1*, when the batch run results create folders under each test called *res1*, those results overwrite the previous *res1* results in the called test's folder.

If you run the batch test without selecting the **Run in batch mode** check box (**Tools** > **General Options** > **Run**), WinRunner saves results only in the subfolder for the batch test. This can cause problems at a later stage if you choose to run the called tests separately, since WinRunner will not know where to look for the previously saved expected and verification results.

When working in unified report mode, all batch run results are saved in a single results folder under the main test's folder.

If a called test calls additional tests, then those results are saved only in the results folder of the test that called it. For example, suppose test A calls tests B and C, test B calls tests D, and E, and test E calls test Z, then when running in batch mode, the results of test B and of test C are stored under test A and also under test B and C, respectively. The results of tests D, E, and Z are all stored only under the main batch test (A) and also under the top-level called test (B).

Viewing Batch Test Results

When a batch test run is completed, you can view information about the events that occurred during the run in the Test Results window. If one of the called tests fails, the batch test is marked as failed.

The Test Results window lists all the events that occurred during the batch test run. Each time a test is called, a *call_test* entry is listed. The details of the *call_test* entry indicate whether the **call** statement was successful. Note that even though a **call** statement is successful, the called test itself may fail, based on the usual criteria for a failed test. You can set criteria for a failed test in the **Run** > **Settings** category of the General Options dialog box. For additional information, refer to Chapter 23, "Setting Global Testing Options" in the *Mercury WinRunner Basic Features User's Guide*.

To view the results of the called test, double-click the *call_test* entry. For more information on viewing test results in the Test Results window, refer to Chapter 21, "Analyzing Test Results" in the *Mercury WinRunner Basic Features User's Guide*.

15

Running Tests from the Command Line

You can run tests directly from the Windows command line.

This chapter describes:

- About Running Tests from the Command Line
- Using the Windows Command Line
- Command Line Options

About Running Tests from the Command Line

You can use the Windows Run command to start WinRunner and run a test according to predefined options. You can also save your startup options by creating a custom WinRunner shortcut. Then, to start WinRunner with the startup options, you simply double-click the icon.

Using the command line, you can:

- start your application
- start WinRunner
- load the relevant tests
- run the tests
- specify test options
- specify the results directories for the test

Most of the functional options that you can set within WinRunner can also be set from the command line. These include test run options and the directories in which test results are stored.

You can also specify a *custom.ini* file that contains these and other environment variables and system parameters.

For example, the following command starts WinRunner, loads a batch test, and runs the test in **Verify** mode:

```
C:\Program Files\Mercury Interactive\WinRunner\WRUN.EXE -t
c:\batch\newclock -batch on -verify -run_minimized -dont_quit -run
```

The test *newclock* is loaded and then executed in batch mode with WinRunner minimized. WinRunner remains open after the test run is completed.

Note: You can use AT commands (specifically the SU.EXE command) with WinRunner. AT commands are part of the Microsoft Windows NT operating system. You can find information on AT commands in the NT Resource Kit. This enables running completely automated scripts, without user intervention.

Using the Windows Command Line

You can use the Windows command line to start WinRunner with predefined options. If you plan to use the same set of options each time you start WinRunner, you can create a custom WinRunner shortcut.

Starting WinRunner from the Command Line

This procedure describes how to start WinRunner from the command line.

To start WinRunner from the Run command:

1. On the Windows **Start** menu, choose **Run**. The Run dialog box opens.
2. Type in the path of your WinRunner *wrun.exe* file, and then type in any command line options you want to use.
3. Click **OK** to close the dialog box and start WinRunner.

Note: If you add command line options to a path containing spaces, you must specify the path of the wrun.exe within quotes, for example:

```
"D:\Program Files\Mercury Interactive\WinRunner\arch\wrun.exe" -addins
WebTest
```

Adding a Custom WinRunner Shortcut

You can make the options you defined permanent by creating a custom WinRunner shortcut.

To add a custom WinRunner shortcut:

1. Create a shortcut for your *wrun.exe* file in Windows Explorer or My Computer.
2. Click the right mouse button on the shortcut and choose **Properties**.
3. Click the **Shortcut** tab.
4. In the **Target** box, type in any command line options you want to use after the path of your WinRunner *wrun.exe* file.
5. Click **OK**.

Command Line Options

Following is a description of each command line option.

-addins *list of add-ins to load*

Instructs WinRunner to load the specified add-ins. In the list, separate the add-ins by commas (without spaces). This can be used in conjunction with the **-addins_select_timeout** command line option.

(Formerly **-addons.**)

Note: All installed add-ins are listed in the registry under: *HKEY_LOCAL_MACHINE\SOFTWARE\Mercury Interactive\WinRunner\CurrentVersion\Installed Components\.*

Use the syntax (spelling) displayed in the key names under this branch when specifying the add-ins to load. The names of the add-ins are not case sensitive.

For example, the following line will load the four add-ins that are included with WinRunner:

```
<WinRunner folder>\arch\wrun.exe -addins ActiveX,pb,vb,WebTest
```

-addins_select_timeout *timeout*

Instructs WinRunner to wait the specified time (in seconds) before closing the **Add-In Manager** dialog box when starting WinRunner. When the timeout is zero, the dialog box is not displayed. This can be used in conjunction with the **-addins** command line option.

(Formerly **-addons_select_timeout.**)

-animate

Instructs WinRunner to execute and run the loaded test, while the execution arrow displays the line of the test being run.

-app *path*

Runs the specified application before running WinRunner. This can be used in conjunction with the **-app_params, -app_open_win,** and **-WR_wait_time** command line options.

Note that you can also define a startup application in the **Run** tab of the Test Properties dialog box. For more information, refer to Chapter 22, "Setting Properties for a Single Test" in the *Mercury WinRunner Basic Features User's Guide*.

-app_params *param1[,param2,...,paramN]*

Passes the specified parameters to the application specified in **-app.**

Note: You can only use this command line option when you also use the **-app** command line option.

-app_open_win *setting*

Determines how the application window appears when it opens.

The following are the possible values for *setting*:

Option	Description
SW_HIDE	Hides the window and activates another window.
SW_SHOWNORMAL	Activates and displays a window. If the window is minimized or maximized, Windows restores it to its original size and position. Specify this flag when displaying the window for the first time.
SW_SHOWMINIMIZED	Activates the window and displays it as a minimized window.
SW_SHOWMAXIMIZED	Activates the window and displays it as a maximized window.
SW_SHOWNOACTIVATE	Displays a window in its most recent size and position. The active window remains active.

Option	Description
SW_SHOW	Activates the window and displays it in its current size and position.
SW_MINIMIZE	Maximizes the specified window and activates the next top-level window in the z-order.
SW_SHOWMINNOACTIVE	Displays the window as a minimized window. The active window remains active.
SW_SHOWNA	Displays the window in its current state. The active window remains active.
SW_RESTORE	Activates and displays the window. If the window is minimized or maximized, Windows restores it to its original size and position. Specify this flag when restoring a minimized window.

Note: You can only use this command line option when you also use the **-app** command line option.

-auto_load {on | off}

Activates or deactivates automatic loading of the temporary GUI map file.

(Default = **on**)

-auto_load_dir *path*

Determines the folder in which the temporary GUI map file (*temp.gui*) resides. This option is applicable only when auto load is on.

(Default = **M_Home\dat**)

-batch {on | off}

Runs the loaded test in Batch mode.

(Default = **off**)

You can also set this option using the **Run in batch mode** check box in the **Run** category of the General Options dialog box as described in Chapter 23, "Setting Global Testing Options" in the *Mercury WinRunner Basic Features User's Guide*.

Note that you can use the **getvar** function to retrieve the value of the corresponding *batch* testing option from within a test script, as described in Chapter 21, "Setting Testing Options from a Test Script."

Tip: To ensure that the test run does not pause to display error messages, use the **-batch** option in conjunction with the **-verify** option. For more information on the **-verify** option, see page 221.

-beep {on | off}

Activates or deactivates the WinRunner system beep.

You can also set this option using the corresponding **Beep when checking a window** check box in the **Run** > **Settings** category of the General Options dialog box, described in Chapter 23, "Setting Global Testing Options" in the *Mercury WinRunner Basic Features User's Guide*.

Note that you can use the **setvar** and **getvar** functions to set and retrieve the value of the corresponding *beep* testing option from within a test script, as described in Chapter 21, "Setting Testing Options from a Test Script."

-capture_bitmap {on | off}

Determines whether WinRunner captures a bitmap whenever a checkpoint fails. When this option is on (1), WinRunner uses the settings from the **Run** > **Settings** category of the General Options dialog box to determine the captured area for the bitmaps.

(Default = **off**)

You can also set this option using the **Capture bitmap on verification failure** check box in the **Run** > **Settings** category of the General Options dialog box, as described in Chapter 23, "Setting Global Testing Options" in the *Mercury WinRunner Basic Features User's Guide*.

Note that you can use the **setvar** and **getvar** functions to set and retrieve the value of the corresponding *capture_bitmap* testing option from within a test script, as described in Chapter 21, “Setting Testing Options from a Test Script.”

-create_text_report {on | off}

Instructs WinRunner to write test results to a text report, *report.txt*, which is saved in the results folder.

-create_unirep_info {on | off}

Generates the necessary information for creating a Unified Report (when WinRunner report view is selected) so that you can choose to view the Unified Report of your tests at a later time.

(Default = **on**)

You can also set this option using the corresponding **Create unified report information** option in the **Run** category of the General Options dialog box as described in Chapter 23, “Setting Global Testing Options” in the *Mercury WinRunner Basic Features User’s Guide*.

-cs_fail {on | off}

Determines whether WinRunner fails a test when Context Sensitive errors occur. A Context Sensitive error is the failure of a Context Sensitive statement during a test. Context Sensitive errors are often due to WinRunner’s failure to identify a GUI object.

For example, a Context Sensitive error will occur if you run a test containing a **set_window** statement with the name of a non-existent window. Context Sensitive errors can also occur when window names are ambiguous. For information about Context Sensitive functions, refer to the *TSL Reference*.

(Default = **off**)

You can also set this option using the corresponding **Fail test when Context Sensitive errors occur** check box in the **Run** > **Settings** category of the General Options dialog box, described in Chapter 23, "Setting Global Testing Options" in the *Mercury WinRunner Basic Features User's Guide*.

Note that you can use the **setvar** and **getvar** functions to set and retrieve the value of the corresponding *cs_fail* testing option from within a test script, as described in Chapter 21, "Setting Testing Options from a Test Script."

-cs_run_delay *non-negative integer*

Sets the time (in milliseconds) that WinRunner waits between executing Context Sensitive statements when running a test.

(Default = **0** [milliseconds])

You can also set this option using the corresponding **Delay between execution of CS statements** box in the **Run** > **Synchronization** category of the General Options dialog box, described in Chapter 23, "Setting Global Testing Options" in the *Mercury WinRunner Basic Features User's Guide*.

Note that you can use the **setvar** and **getvar** functions to set and retrieve the value of the corresponding *cs_run_delay* testing option from within a test script, as described in Chapter 21, "Setting Testing Options from a Test Script."

-def_replay_mode {verify | debug | update}

Sets the run mode that is used by default for all tests.

Note: Verify mode is only relevant when running tests, not components. When working with components, the application is verified when the component is run as part of a business process test in Quality Center.

Possible values:

- **Update—**Used to update the expected results of a test or to create a new expected results folder.
- **Verify—**Used to check your application.
- **Debug—**Used to help you identify bugs in a test script.

(Default = **Verify**)

You can also set this option using the **Default run mode** option in the **Run** category of the General Options dialog box as described in Chapter 23, “Setting Global Testing Options” in the *Mercury WinRunner Basic Features User’s Guide*.

-delay_msec *non-negative integer*

Directs WinRunner to determine whether a window or object is stable before capturing it for a bitmap checkpoint or synchronization point. It defines the time (in milliseconds) that WinRunner waits between consecutive samplings of the screen. If two consecutive checks produce the same results, WinRunner captures the window or object. (Formerly **-delay**, which was measured in seconds.)

(Default = **1000** [milliseconds])

(Formerly **-delay**.)

Note: This parameter is accurate to within 20-30 milliseconds.

You can also set this option using the corresponding **Delay for window synchronization** box in the **Run** > **Synchronization** category of the General Options dialog box, described in Chapter 23, "Setting Global Testing Options" in the *Mercury WinRunner Basic Features User's Guide*.

Note that you can use the **setvar** and **getvar** functions to set and retrieve the value of the corresponding *delay_msec* testing option from within a test script, as described in Chapter 21, "Setting Testing Options from a Test Script."

-dont_connect

If the **Reconnect on startup** option is selected in the Connection to Quality Centerdialog box, this command line enables you to open WinRunner without connecting to Test Director.

To disable the **Reconnect on startup** option, choose **Tools** > **Quality Center Connection** and clear the **Reconnect on startup c**heck box as described in Chapter 15, "Running Tests from the Command Line".

-dont_quit

Instructs WinRunner not to close after completing the test.

-dont_show_welcome

Instructs WinRunner not to display the Welcome window when starting WinRunner.

-email_service

Determines whether WinRunner activates the e-mail sending options including the e-mail notifications for checkpoint failures, test failures, and test completed reports as well as any **email_send_msg** statements in the test.

(Default = **off**)

You can also set this option using the corresponding Activate e-mail service check box in the **Notifications** > **E-mail** category of the General Options dialog box as described in Chapter 23, "Setting Global Testing Options" in the *Mercury WinRunner Basic Features User's Guide*.

Note that you can use the **setvar** and **getvar** functions to set and retrieve the value of the corresponding **email_service** testing option from within a test script, as described in Chapter 21, "Setting Testing Options from a Test Script."

-exp *expected results folder name*

Designates a name for the subfolder in which expected results are stored. In a verification run, specifies the set of expected results used as the basis for the verification comparison.

(Default = **exp**)

You can also view this setting using the corresponding **Expected results folder** box in the **Current Test** tab of the Test Properties dialog box, described in Chapter 22, "Setting Properties for a Single Test" in the *Mercury WinRunner Basic Features User's Guide*.

Note that you can use the **getvar** function to retrieve the value of the corresponding *exp* testing option from within a test script, as described in Chapter 21, "Setting Testing Options from a Test Script."

-fast_replay {on | off}

Sets the speed of the test run for tests recorded in Analog mode. **on** sets tests to run as fast as possible and **off** sets tests to run at the speed at which they were recorded.

Note that you can also specify the analog run speed using the **Run speed for Analog mode** option in the **Run** category of the General Options dialog box as described in Chapter 23, "Setting Global Testing Options" in the *Mercury WinRunner Basic Features User's Guide*.

(Default = **on**)

-f *file name*

Specifies a text file containing command line options. The options can appear on the same line, or each on a separate line. This option enables you to circumvent the restriction on the number of characters that can be typed into the Target text box in the **Shortcut** tab of the Windows Properties dialog box.

Note: If a command line option appears both in the command line and in the file, WinRunner uses the settings of the option in the file.

-fontgrp *group name*

Specifies the active font group when WinRunner is started.

You can also set this option using the corresponding **Font group** box in the **Record** > **Text Recognition** category of the General Options dialog box, described in Chapter 23, "Setting Global Testing Options" in the *Mercury WinRunner Basic Features User's Guide*.

Note that you can use the **setvar** and **getvar** functions to set and retrieve the value of the corresponding *fontgrp* testing option from within a test script, as described in Chapter 21, "Setting Testing Options from a Test Script."

-ini *pathname* wrun.ini file

Defines the *wrun.ini* file that is used when WinRunner is started. This file is read-only, unless the **-update_ini** command line option is also used.

The path must be a mapped drive, and not a Universal Naming Convention path (e.g. \\<servername>\<sharename>\<directory>).

-min_diff *non-negative integer*

Defines the number of pixels that constitute the threshold for an image mismatch.

(Default = **0** [pixels])

You can also set this option using the corresponding **Threshold for difference between bitmaps** box in the **Run** > **Settings** category of the General Options dialog box, described in Chapter 23, "Setting Global Testing Options" in the *Mercury WinRunner Basic Features User's Guide*.

Note that you can use the **setvar** and **getvar** functions to set and retrieve the value of the corresponding *min_diff* testing option from within a test script, as described in Chapter 21, "Setting Testing Options from a Test Script."

-mismatch_break {on | off}

Activates or deactivates Break when Verification Fails before a verification run. The functionality of Break when Verification Fails is different than when running a test interactively: In an interactive run, the test is paused; For a test started from the command line, the first occurrence of a comparison mismatch terminates the test run.

Break when Verification Fails determines whether WinRunner pauses the test run and displays a message whenever verification fails or whenever any message is generated as a result of a Context Sensitive statement during a test that is run in **Verify** mode.

For example, if a **set_window** statement is missing from a test script, WinRunner cannot find the specified window. If this option is on, WinRunner pauses the test and opens the Run wizard to enable the user to locate the window. If this option is off, WinRunner reports an error in the Test Results window and proceeds to run the next statement in the test script.

(Default = **on**)

You can also set this option using the corresponding **Break when verification fails** check box in the **Run** > **Settings** category of the General Options dialog box, described in Chapter 23, "Setting Global Testing Options" in the *Mercury WinRunner Basic Features User's Guide*.

Note that you can use the **setvar** and **getvar** functions to set and retrieve the value of the corresponding *mismatch_break* testing option from within a test script, as described in Chapter 21, "Setting Testing Options from a Test Script."

-qc_connection {on | off}

Activates WinRunner's connection to Quality Center when set to **on**.

(Default = **off**)

(Formerly **-td_connection** or **-test_director.**)

Note that you can connect to Quality Center from the Quality Center Connection dialog box, which you open by choosing **Tools** > **Quality Center Connection**. For more information about connecting to Quality Center, see Chapter 26, "Managing the Testing Process."

Note: If you select the "Reconnect on startup" option in the Connection to Quality Center dialog box, setting **-qc_connection** to off will not prevent the connection to Quality Center. To prevent the connection to Quality Center in this situation, use the **-dont_connect** command. For more information, see "-dont_connect," on page 211.

-qc_cycle_name *cycle name*

Specifies the name of the current test cycle. This option is applicable only when WinRunner is connected to Quality Center.

Note that you can use the corresponding *qc_cycle_name* testing option to specify the name of the current test cycle, as described in Chapter 21, "Setting Testing Options from a Test Script."

(Formerly **-td_cycle_name** or **-cycle.**)

-qc_database_name *database path*

Specifies the active Quality Center database. WinRunner can open, execute, and save tests in this database. This option is applicable only when WinRunner is connected to Quality Center.

Use the following syntax when using this option:

```
<database_name>.<domain>
```

For example:

```
Mercury.Wrun
```

Note that you can use the corresponding *qc_database_name* testing option to specify the active Quality Center database, as described in Chapter 21, “Setting Testing Options from a Test Script.”

Note that when WinRunner is connected to Quality Center, you can specify the active Quality Center project database from the Quality Center Connection dialog box, which you open by choosing **Tools** > **Quality Center Connection**. For more information, see Chapter 26, “Managing the Testing Process.”

(Formerly **-td_database_name** or **-database.**)

-qc_password *password*

Specifies the password for connecting to a database in a Quality Center server.

Note that you can specify the password for connecting to Quality Center from the Quality Center Connection dialog box, which you open by choosing **Tools** > **Quality Center Connection**. For more information about connecting to Quality Center, see Chapter 26, “Managing the Testing Process.”

(Formerly **-td_password**)

-qc_server_name *server name*

Specifies the name of the Quality Center server to which WinRunner connects.

Note that you can use the corresponding *qc_server_name* testing option to specify the name of the Quality Center server to which WinRunner connects, as described in Chapter 21, “Setting Testing Options from a Test Script.”

In order to connect to the server, use the **td_connection** option.

(Formerly **-td_server_name** or **-td_server.**)

-qc_user_name ***user name***

Specifies the name of the user who is currently executing a test cycle.

Note that you can use the corresponding *qc_user_name* testing option to specify the user, as described in Chapter 21, "Setting Testing Options from a Test Script."

Note that you can specify the user name when you connect to Quality Center from the Quality Center Connection dialog box, which you open by choosing **Tools** > **Quality Center Connection**. For more information about connecting to Quality Center, see Chapter 26, "Managing the Testing Process."

(Formerly **-td_user_name**, **-user_name,** or **-user.**)

-rec_item_name {0 | 1}

Determines whether WinRunner records non-unique ListBox and ComboBox items by name or by index.

(Default = **0**)

You can also set this option using the corresponding **Record non-unique list items by name** check box in the **Record** category of the General Options dialog box, described in Chapter 23, "Setting Global Testing Options" in the *Mercury WinRunner Basic Features User's Guide*.

Note that you can use the **setvar** and **getvar** functions to set and retrieve the value of the corresponding *rec_item_name* testing option from within a test script, as described in Chapter 21, "Setting Testing Options from a Test Script."

-run

Instructs WinRunner to run the loaded test. To load a test into the WinRunner window, use the **-t** command line option.

-run_minimized

Instructs WinRunner to open and run tests with WinRunner and the test minimized to an icon. Note that specifying this option does not itself run tests: use the **-t** command line option to load a test and the **-run** command line option to run the loaded test.

-search_path *path*

Defines the directories to be searched for tests to be opened and/or called. The search path is given as a string.

(Default = **startup folder** and **installation folder\lib**)

You can also set this option using the corresponding **Search path for called tests** box in the **Folders** category of the General Options dialog box, described in Chapter 23, “Setting Global Testing Options” in the *Mercury WinRunner Basic Features User’s Guide*.

Note that you can use the **setvar** and **getvar** functions to set and retrieve the value of the corresponding *searchpath* testing option from within a test script, as described in Chapter 21, “Setting Testing Options from a Test Script.”

-single_prop_check_fail {on | off}

Fails a test run when **_check_info** statements fail. It also writes an event to the Test Results window for these statements. (You can create **_check_info** statements using the **Insert** > **GUI Checkpoint** > **For Single Property** command.)

You can use this option with the **setvar** and **getvar** functions.

(Default = **on**)

For information about the **check_info** functions, refer to the *TSL Reference*.

You can also set this option using the corresponding **Fail test when single property check fails** option in the **Run** > **Settings** category of the General Options dialog box, described in Chapter 23, “Setting Global Testing Options” in the *Mercury WinRunner Basic Features User’s Guide*.

Note that you can use the **setvar** and **getvar** functions to set and retrieve the value of the corresponding *single_prop_check_fail* testing option from within a test script, as described in Chapter 21, “Setting Testing Options from a Test Script.”

-speed {normal | fast}

Sets the speed for the execution of the loaded test.

(Default = **fast**)

You can also set this option using the corresponding **Run Speed for Analog Mode** option in the **Run** category of the General Options dialog box, described in Chapter 23, "Setting Global Testing Options" in the *Mercury WinRunner Basic Features User's Guide*.

Note that you can use the **setvar** and **getvar** functions to set and retrieve the value of the corresponding *speed* testing option from within a test script, as described in Chapter 21, "Setting Testing Options from a Test Script."

(Formerly **-run_speed.**)

-start_minimized {on | off}

Indicates whether WinRunner opens in minimized mode.

(Default = **off**)

-t *test name*

Specifies the name of the test to be loaded in the WinRunner window. This can be the name of a test stored in a folder specified in the search path or the full pathname of any test stored in your system.

-timeout_msec *non-negative integer*

Sets the global timeout (in milliseconds) used by WinRunner when executing checkpoints and Context Sensitive statements. This value is added to the *time* parameter embedded in GUI checkpoint or synchronization point statements to determine the maximum amount of time that WinRunner searches for the specified window or object. (Formerly *timeout*, which was measured in seconds.)

(Default = **10,000** [milliseconds])

(Formerly **-timeout**.)

Note: This option is accurate to within 20-30 milliseconds.

You can also set this option using the corresponding **Timeout for checkpoints and CS statements** box in the **Run** > **Settings** category of the General Options dialog box, described in Chapter 23, "Setting Global Testing Options" in the *Mercury WinRunner Basic Features User's Guide*.

Note that you can use the **setvar** and **getvar** functions to set and retrieve the value of the corresponding *timeout_msec* testing option from within a test script, as described in Chapter 21, "Setting Testing Options from a Test Script."

-tslinit_exp *expected results folder*

Directs WinRunner to the expected folder to be used when the *tslinit* script is running.

-update_ini

Saves changes to configuration made during a WinRunner session when the *wrun.ini* file is specified by the **-ini** command line option.

Note: You can only use this command line option when you also use the **-ini** command line option.

-verify *verification results folder name*

Specifies that the test is to be run in **Verify** mode and designates the name of the subfolder in which the test results are stored.

-WR_wait_time *non-negative integer*

Specifies the number of milliseconds to wait between invoking the application and starting WinRunner.

(Default = **0** [milliseconds])

You can also set this option using the **Run test after** box in the **Run** tab of the Test Properties dialog box, described in Chapter 22, “Setting Properties for a Single Test” in the *Mercury WinRunner Basic Features User’s Guide*.

Note: You can only use this command line option when you also use the **-app** command line option.

Part V

Debugging Tests

16

Controlling Your Test Run

Controlling the test run can help you to identify and eliminate defects in your test scripts.

This chapter describes:

- About Controlling Your Test Run
- Running a Single Line of a Test Script
- Running a Section of a Test Script
- Pausing a Test Run

About Controlling Your Test Run

After you create a test script you should check that it runs smoothly, without errors in syntax or logic. In order to detect and isolate defects in a script, you can use the Step and Pause commands to control test execution.

The following Step commands are available:

- The Step command runs a single line of a test script.
- The Step Into command calls and displays another test or user-defined function.
- The Step Out command—used in conjunction with Step Into—completes the execution of a called test or user-defined function.
- The *S*tep to Cursor command runs a selected section of a test script.

In addition, you can use the Pause command or the **pause** function to temporarily suspend the test run.

You can also control the test run by setting breakpoints. A breakpoint pauses a test run at a pre-determined point, enabling you to examine the effects of the test on your application. You can view all breakpoints in the Breakpoints List pane of the Debug Viewer. For more information, see Chapter 17, “Using Breakpoints.”

To help you debug your tests, WinRunner enables you to monitor variables in a test 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. You can view the current values of monitored variables in the Watch List pane of the Debug Viewer. For more information, see Chapter 18, “Monitoring Variables.”

You can use the call chain to follow and navigate the test flow. At each break during a test run—such as after a Step command, at a breakpoint, or at the end of a test, you can view the current chain of called tests and functions in the Call Chain pane of the Debug Viewer. For more information, see Chapter 9, “Calling Tests.”

When you debug a test script, you run the test in **Debug** mode. The results of the test are saved in a *debug* folder. Each time you run the test, the previous debug results are overwritten. Continue to run the test in **Debug** mode until you are ready to run it in **Verify** mode. For more information on using the **Debug** mode, refer to Chapter 20, “Understanding Test Runs” in the *Mercury WinRunner Basic Features User’s Guide*.

Running a Single Line of a Test Script

You can run a single line of a test script using the Step, Step Into, and Step Out commands.

Step

Choose the **Step** command or click the corresponding **Step** button to execute only the current line of the active test script—the line marked by the execution arrow.

When the current line calls another test or a user-defined function, the called test or function is executed in its entirety but the called test script is not displayed in the WinRunner window. If you are using a startup application or startup function, it is also executed.

Step Into

Choose the **Step Into** command or click the corresponding **Step Into** button to execute only the current line of the active test script. However, in contrast to Step, if the current line of the executed test calls another test or a user-defined function in compiled mode:

- The test script of the called test or function is displayed in the WinRunner window.
- Startup application and function settings (Test Properties dialog box, **Run** tab) are not implemented.
- Use Step or Step Out to continue running the called test.

Step Out

You use the **Step Out** command only after entering a test or a user-defined function using Step Into. Step Out executes to the end of the called test or user-defined function, returns to the calling test, and then pauses the test run.

Running a Section of a Test Script

You can execute a selected section of a test script using the Step to Cursor command.

To use the Step to Cursor command:

1. Move the execution arrow to the line in the test script from which you want to begin test execution. To move the arrow, click inside the margin next to the desired line in the test script.
2. Click inside the test script to move the cursor to the line where you want test execution to stop.
3. Choose **Debug >Step to Cursor** or press the STEP TO CURSOR softkey. WinRunner runs the test up to the line marked by the insertion point.

Pausing a Test Run

You can temporarily suspend a test run by choosing the Pause command or by adding a **pause** statement to your test script.

Pause Command

You can suspend the running of a test by choosing **Test** > **Pause**, clicking the **Pause** button, or pressing the PAUSE softkey. A paused test stops running when all previously interpreted TSL statements have been executed. Unlike the **Stop** command, **Pause** does not initialize test variables and arrays.

To resume running of a paused test, choose the appropriate Run command on the **Test** menu. The test run continues from the point that you invoked the Pause command, or from the execution arrow if you moved it while the test was suspended.

The pause Function

When WinRunner processes a **pause** statement in a test script, test execution halts and a message box is displayed. If the **pause** statement includes an expression, the result of the expression appears in the message box. The syntax of the **pause** function is:

pause ([*expression*] **);**

In the following example, **pause** suspends the test run and displays the time that elapsed between two points.

```
t1=get_time();
t2=get_time();
pause ("Time elapsed" is & t2-t1);
```

Note: The **pause** statement is ignored by WinRunner when running tests in batch mode.

For more information on the **pause** function, refer to the *TSL Reference*.

17

Using Breakpoints

A breakpoint marks a place in the test script where you want to pause a test run. Breakpoints help to identify flaws in a script.

This chapter describes:

- About Using Breakpoints
- Choosing a Breakpoint Type
- Setting Break at Location Breakpoints
- Setting Break in Function Breakpoints
- Modifying Breakpoints
- Deleting Breakpoints

About Using Breakpoints

By setting a breakpoint you can stop a test run at a specific place in a test script. A breakpoint is indicated by a breakpoint marker in the left margin of the test window.

WinRunner pauses the test run when it reaches a breakpoint. You can examine the effects of the test run up to the breakpoint, view the current value of variables, make any necessary changes, and then continue running the test from the breakpoint. You use the **Run from Arrow** command to restart the test run from the breakpoint. Once restarted, WinRunner continues running the test until it encounters the next breakpoint or the test is completed.

Note: WinRunner only pauses when it is not in batch mode. When running tests in batch mode, WinRUnner ignores breakpoints.

Breakpoints are useful for:

- suspending the test run at a certain point and inspecting the state of your application.
- monitoring the entries in the Watch List. See Chapter 18, "Monitoring Variables," for more information.
- marking a point from which to begin stepping through a test script using the Step commands. See Chapter 16, "Controlling Your Test Run," for more information.

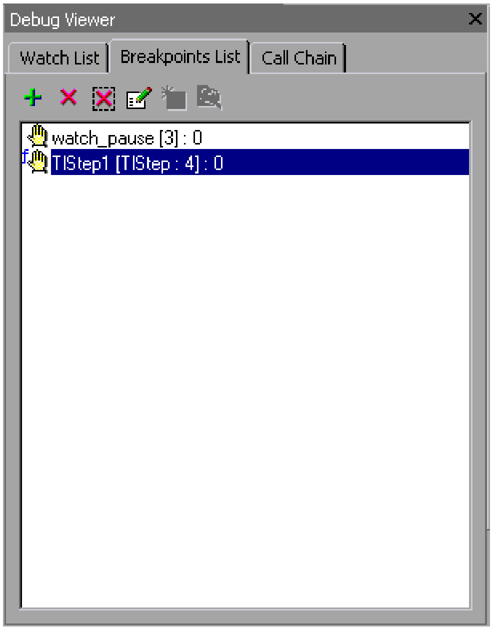
There are two types of breakpoints: Break at Location and Break in Function. A Break at Location breakpoint stops a test at a specified line number in a test script. A Break in Function breakpoint stops a test when it calls a specified user-defined function in a loaded compiled module.

You set a pass count for each breakpoint you define. The pass count determines the number of times the breakpoint is passed before it stops the test run. For example, suppose you program a loop that performs a command twenty-five times. By default, the pass count is set to zero, so test execution stops after each loop. If you set the pass count to 25, execution stops only after the twenty-fifth iteration of the loop.

Note: The breakpoints you define are active only during your current WinRunner session. If you terminate your WinRunner session, you must redefine breakpoints to continue debugging the script in another session.

Viewing the Breakpoints List in the Debug Viewer

You view the values of variables in the Breakpoints List pane in the Debug Viewer window. If the Debug Viewer window is not currently displayed, or the Breakpoints List pane is not open in the window, choose **Debug > Breakpoints List** to display it. If the Breakpoints List pane is open, but a different pane is currently displayed, click the **Breakpoints List** tab to display it.



watch_pause [3] : 0
TlStep1 [TlStep : 4] : 0

Tip: The Debug Viewer window can be displayed as a docked window within the WinRunner window, or it can be a floating window that you can drag to any location on your screen. By default the Debug Viewer opens as a docked window on the right side of the WinRunner screen. To move the window to another location, drag the Debug Viewer titlebar.

Choosing a Breakpoint Type

WinRunner enables you to set two types of breakpoints: Break at Location and Break in Function.

Break at Location

A Break at Location breakpoint stops a test at a specified line number in a test script. This type of breakpoint is defined by a test name and a test script line number. The breakpoint marker appears in the left margin of the test script, next to the specified line. A Break at Location breakpoint might, for example, appear in the Breakpoints List pane as:

```
ui_test[137] : 0
```

This means that the breakpoint marker appears in the test named *ui_test* at line 137. The number after the colon represents the pass count, which is set here to zero (the default). This means that WinRunner will stop running the test every time it passes the breakpoint.

Break in Function

A Break in Function breakpoint stops a test when it calls a specified user-defined function in a loaded compiled module. This type of breakpoint is defined by the name of a user-defined function and the name of the compiled module in which the function is located. When you define a Break in Function breakpoint, the breakpoint marker appears in the left margin of the WinRunner window, next to the first line of the function. WinRunner halts the test run each time the specified function is called. A Break in Function breakpoint might appear in the Breakpoints List pane as:

```
ui_func [ui_test : 25] : 10
```

This indicates that a breakpoint has been defined for the line containing the *ui_func* function, in the *ui_test* compiled module: in this case line 25. The pass count is set to 10, meaning that WinRunner stops the test each time the function has been called ten times.

Setting Break at Location Breakpoints

You set Break at Location breakpoints using the Breakpoints List pane in the Debug Viewer, the mouse, or the Toggle Breakpoint command.

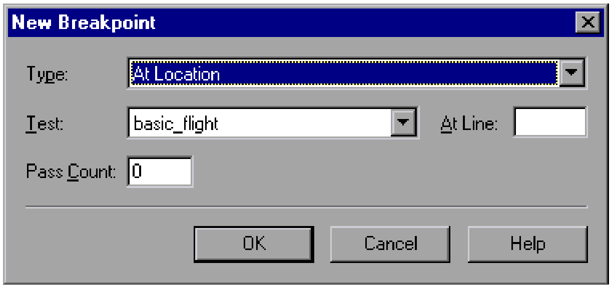
Note: You can set a breakpoint in a function only after the function has been loaded into WinRunner (the function has been executed at least once).

To set a Break at Location breakpoint using the Breakpoints List pane:

1 Display the Breakpoints List as described in "Viewing the Breakpoints List in the Debug Viewer" on page 233.

2 Click **Add Entry** to open the New Breakpoint dialog box.

3 In the **Type** box, select **At Location**.



New Breakpoint

Type: At Location

Test: basic_flight

At Line:

Pass Count: 0

4 The **Test** box displays the name of the active test. If you want to insert a breakpoint for another test, select the name from the **Test** list.

5 Enter the line number at which you want to add the breakpoint in the **At Line** box

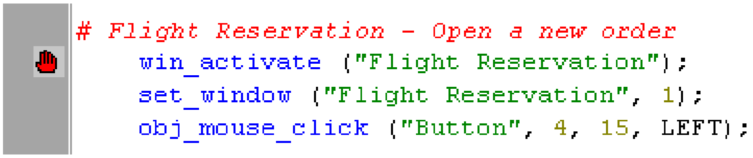
6 If you want the test to break each time it reaches the breakpoint, accept the default **Pass Count**, 0. If you only want the test to break after it reaches the breakpoint a given number of times, enter the number in the **Pass Count** box.

7. Click **OK** to set the breakpoint and close the New Breakpoint dialog box. The new breakpoint is displayed in the Breakpoints List pane.

   The breakpoint marker appears in the left margin of the test script, next to the specified line.

To set a Break at Location breakpoint using the mouse:

1. Right-click the left (gray) margin of the WinRunner window next to the line where you want to add a breakpoint. The breakpoint symbol appears in the left margin of the WinRunner window:



```
# Flight Reservation - Open a new order
    win_activate ("Flight Reservation");
    set_window ("Flight Reservation", 1);
    obj_mouse_click ("Button", 4, 15, LEFT);
```

Tip: If the gray margin is not visible, choose **Tools** > **Editor Options** and click the **Options** tab. Then select the **Visible gutter** option.

2. Breakpoints added using this method automatically use a pass count of 0. If you want to use a different pass count, modify the breakpoint as described in “Modifying Breakpoints” on page 239.

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

1. Move the insertion point to the line of the test script where you want test execution to stop.
2. Choose **Debug** > **Toggle Breakpoint** or click the **Toggle Breakpoint** button. The breakpoint symbol appears in the left margin of the WinRunner window and is displayed in the Breakpoints List.
3. Breakpoints added using this method automatically use a pass count of 0. If you want to use a different pass count, modify the breakpoint as described in “Modifying Breakpoints” on page 239.

To remove a Break at Location breakpoint:

Right-click the breakpoint symbol

or:

Choose **Debug** > **Toggle Breakpoint**, or click the **Toggle Breakpoint** button.

Setting Break in Function Breakpoints

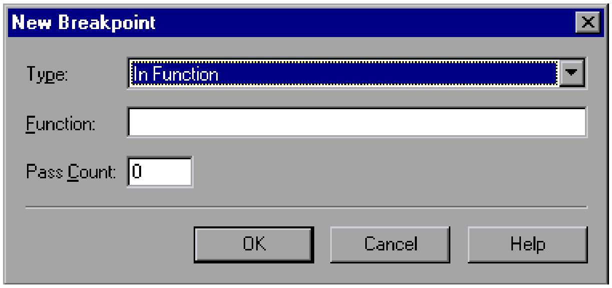
A Break in Function breakpoint stops test execution at the user-defined function that you specify. You set a Break in Function breakpoint from the Breakpoint Lists pane in the Debug Viewer, or the **Break in Function** command.

Note: You can set a breakpoint in a function only after the function has been loaded into WinRunner (the function has been executed at least once).

To set a Break in Function breakpoint:

1. If you want to set a break in function breakpoint for a function that is already a part of your test, place the insertion point on the function name.
2. Choose **Debug** > **Break in Function**. The New Breakpoint dialog box opens. Proceed to step 5.
3. Alternatively, you can open the New Breakpoint dialog box from the Breakpoint Lists pane. Display the Breakpoints List as described in “Viewing the Breakpoints List in the Debug Viewer” on page 233.
4. Click **Add Entry.**

5 The New Breakpoint dialog box opens.



New Breakpoint

Type: In Function

Function:

Pass Count: 0

Accept the breakpoint type: **In Function**.

6 By default, the **Function** box displays the name of the function (or text) in which the insertion point is currently located. Accept the function name or enter the name of a valid function. The function name you specify must be compiled by WinRunner. For more information, see Chapter 10, "Creating User-Defined Functions," and Chapter 11, "Employing User-Defined Functions in Tests."

7 Type a value in the **Pass Count** box.

8 Click **OK** to set the breakpoint and close the New Breakpoint dialog box.

The new breakpoint is displayed in the Breakpoints List pane.

The breakpoint symbol is displayed in the left margin next to the first line of the function in the compiled module.

Modifying Breakpoints

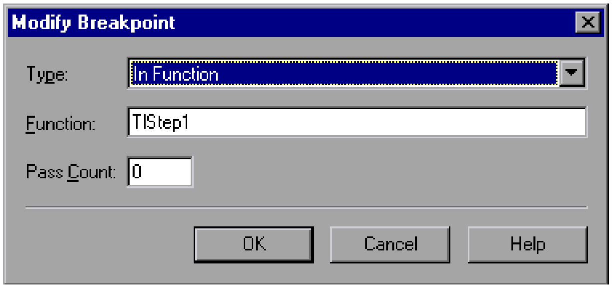
You can modify the definition of a breakpoint using the Modify Breakpoints dialog box. You can change the breakpoint's type, the test or line number for which it is defined, and the value of the pass count.

To modify a breakpoint:

1 Display the Breakpoints List as described in "Viewing the Breakpoints List in the Debug Viewer" on page 233.

2 Select a breakpoint in Breakpoint Lists pane.

3 Click **Modify entry** to open the Modify Breakpoint dialog box.



Modify Breakpoint

Type: In Function

Function: TIStep1

Pass Count: 0

4 To change the type of breakpoint, select a different breakpoint type in the **Type** box.

5 Change the settings as necessary.

6 Click **OK** to close the dialog box.

Deleting Breakpoints

You can delete a single breakpoint or all breakpoints defined for the current test using the Breakpoints dialog box.

To delete a single breakpoint:

1. Display the Breakpoints List as described in “Viewing the Breakpoints List in the Debug Viewer” on page 233.
2. Select a breakpoint from the list.
3. Click **Delete entry**. The breakpoint is removed from the list and the breakpoint symbol is removed from the left margin of the test.

To delete all breakpoints using the Delete All Breakpoints Command:

Choose **Debug** > **Delete All Breakpoints** or click the **Delete All Breakpoints** toolbar button.

To delete all breakpoints using the Debug Viewer:

1. Display the Breakpoints List as described in “Viewing the Breakpoints List in the Debug Viewer” on page 233.
2. Click **Delete all breakpoints**. All breakpoints are deleted from the list and all breakpoint symbols are removed from the left margin of the relevant tests.

18

Monitoring Variables

The Watch List displays the values of variables, expressions, and array elements during a test run. You use the Watch List to enhance the debugging process.

This chapter describes:

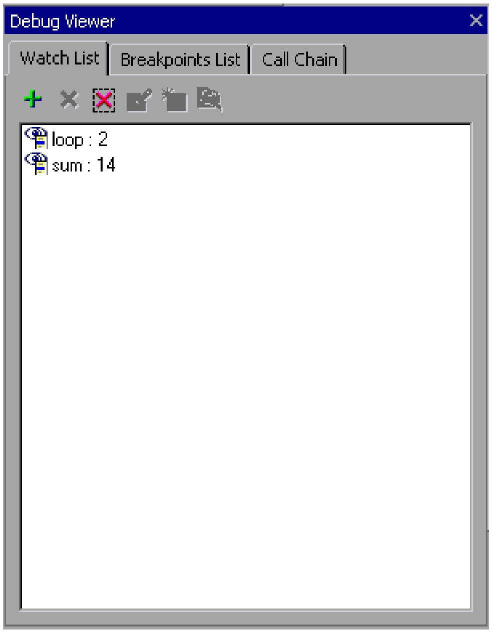
- About Monitoring Variables
- Adding Variables to the Watch List
- Viewing Variables in the Watch List
- Modifying Variables in the Watch List
- Assigning a Value to a Variable in the Watch List
- Deleting Variables from the Watch List

About Monitoring Variables

The Watch List enables you to monitor the values of variables, expressions, and array elements while you debug a test script. Prior to running a test, you add the elements that you want to monitor to the Watch List. At each break during a test run—such as after a Step command, at a breakpoint, or at the end of a test, you can view the current values of the entries in the Watch List.

Viewing the Watch List in the Debug Viewer

You view the values of variables in the Watch List pane in the Debug Viewer window. If the Debug Viewer window is not currently displayed, or the Watch List pane is not open in the window, choose **Debug** > **Watch List** to display it. If the Watch List pane is open, but a different pane is currently displayed, click the **Watch List** tab to display it.

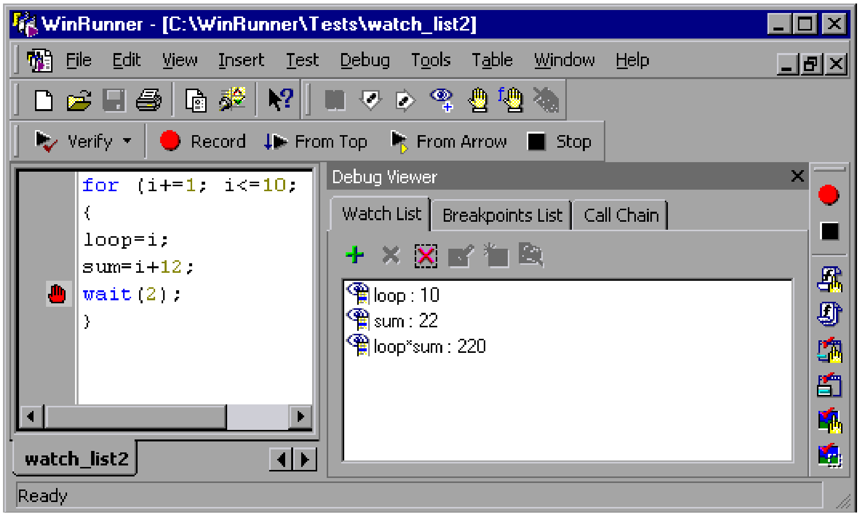


loop : 2
sum : 14

Tip: The Debug Viewer window can be displayed as a docked window within the WinRunner window, or it can be a floating window that you can drag to any location on your screen. By default the Debug Viewer opens as a docked window on the right side of the WinRunner screen. To move the window to another location, drag the Debug Viewer titlebar.

Watching Variable Values—An Example

For example, in the following test, the Watch List is used to measure and track the values of variables *loop (*the current loop*)* and *sum*. On the last step of each loop, the test pauses at the breakpoint so you can view the current values.



```
for (i+=1; i<=10;
{
loop=i;
sum=i+12;
wait(2);
}
```

```
loop : 10
sum : 22
loop*sum : 220
```

After WinRunner executes the first loop, the test pauses. The Watch List displays the variables and updates their values: When WinRunner completes the test run, the Watch List shows the following results:

```
loop:10
sum:22
loop*sum:220
```

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 is the current value for the test that WinRunner is interpreting.

If you choose a test or function in the Call Chain list (**Debug** > **Call Chain**), the context of the variables and expressions in the Watch List changes. WinRunner automatically updates their values in the Watch List.

Adding Variables to the Watch List

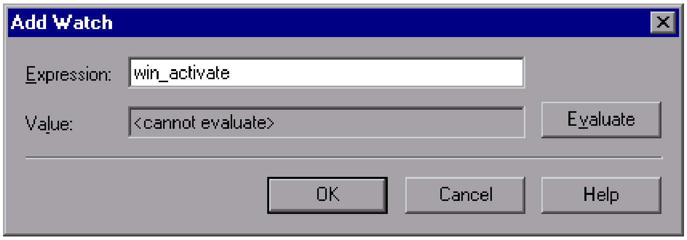
You add variables, expressions, and arrays to the Watch List using the Add Watch dialog box. You can add entries before running a test or when the test breaks after a Step command, when the test is paused, or at a breakpoint.

To add a variable, an expression, or an array to the Watch List:

1 Choose **Debug** > **Add Watch** or click the **Add Watch** button.

Alternatively, display the Watch List as described in "Viewing the Watch List in the Debug Viewer" on page 242 and click **Add entry**.

2 The Add Watch dialog box opens.



Add Watch

Expression: win_activate

Value: <cannot evaluate>

In the **Expression** box, enter the variable, expression, or array that you want to add to the Watch List.

3 Click **Evaluate** to see the current value of the new entry. If the new entry contains a variable or an array that has not yet been initialized, the message "<cannot evaluate>" appears in the **Value** box. The same message appears if you enter an expression that contains an error.

4 Click **OK**. The Add Watch dialog box closes and the new entry appears in the **Watch List**.

Note: Do not add expressions that assign or increment the value of variables to the Watch List; this can affect the test run.

Viewing Variables in the Watch List

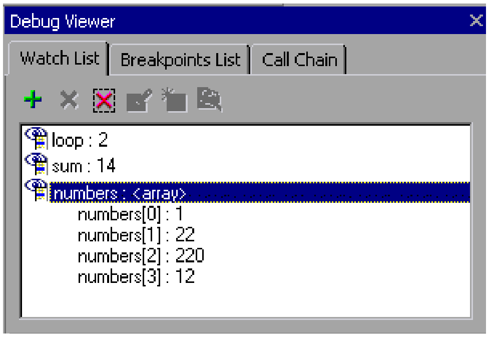
Once you add variables, expressions, and arrays to the Watch List, you can use the Watch List to view their values.

To view the values of variables, expressions, and arrays in the Watch List:

1 Display the Watch List as described in “Viewing the Watch List in the Debug Viewer” on page 242.

The variables, expressions and arrays are displayed; current values appear after the colon.

2 To view values of array elements, double-click the array name. The elements and their values appear under the array name. Double-click the array name to hide the elements.



```
loop : 2
sum : 14
numbers : <array>
    numbers[0] : 1
    numbers[1] : 22
    numbers[2] : 220
    numbers[3] : 12
```

3 Click **Close**.

Modifying Variables in the Watch List

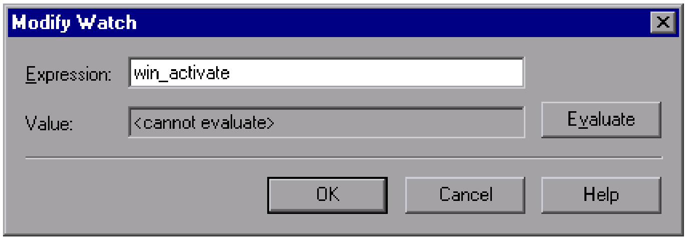
You can modify variables and expressions in the Watch List using the Modify Watch dialog box. For example, you can turn variable *b* into the expression *b + 1*, or you can change the expression *b + 1* into *b * 10*. When you close the Modify Watch dialog box, the Watch List is automatically updated to reflect the new value for the expression.

To modify an expression in the Watch List:

1 Display the Watch List as described in "Viewing the Watch List in the Debug Viewer" on page 242.

2 Select the variable or expression you want to modify.

3 Click **Modify entry** to open the Modify Watch dialog box.



Modify Watch

Expression: win_activate

Value: <cannot evaluate>

4 Change the expression in the **Expression** box as needed.

5 Click **Evaluate**. The new value of the expression appears in the **Value** box.

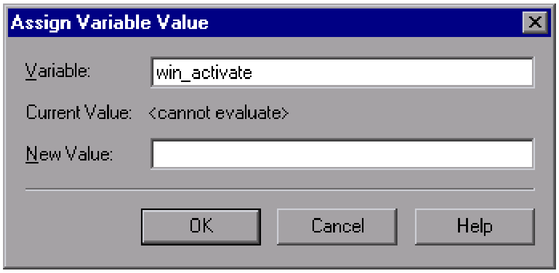
6 Click **OK** to close the Modify Watch dialog box. The modified expression and its new value appear in the Watch List.

Assigning a Value to a Variable in the Watch List

You can assign new values to variables and array elements in the Watch List. Values can be assigned only to variables and array elements, not to expressions.

To assign a value to a variable or an array element:

1. Display the Watch List as described in "Viewing the Watch List in the Debug Viewer" on page 242.
2. Select a variable or an array element.
3. Click **Assign Variable Value** to open the Assign Variable Value dialog box.



Assign Variable Value

Variable: win_activate

Current Value: <cannot evaluate>

New Value:

4. Type the new value for the variable or array element in the **New Value** box.
5. Click **OK** to close the dialog box. The new value appears in the Watch List.

Deleting Variables from the Watch List

You can delete selected variables, expressions, and arrays from the Watch List, or you can delete all the entries in the Watch List.

To delete a variable, an expression, or an array:

1. Display the Watch List as described in “Viewing the Watch List in the Debug Viewer” on page 242.
2. Select a variable, an expression, or an array to delete.

Note: You can delete an array only if its elements are hidden. To hide the elements of an array, double-click the array name in the Watch List.

---

3. Click **Delete entry** to remove the entry from the list.
4. Click **Close** to close the dialog box.

To delete all entries in the Watch List:

1. Display the Watch List as described in “Viewing the Watch List in the Debug Viewer” on page 242.
2. Click **Delete all entries**. All entries are deleted.
3. Click **Close** to close the dialog box.

Part VI

Configuring Advanced Settings

19

Customizing the Test Script Editor

WinRunner includes a powerful and customizable script editor. This enables you to set the size of margins in test windows, change the way the elements of a test script appear, and create a list of typing errors that will be automatically corrected by WinRunner.

This chapter describes:

- About Customizing the Test Script Editor
- Setting Display Options
- Personalizing Editing Commands

About Customizing the Test Script Editor

WinRunner's script editor lets you set display options, and personalize script editing commands.

Setting Display Options

Display options let you configure WinRunner's test windows and how your test scripts are displayed. For example, you can set the size of test window margins, and activate or deactivate word wrapping.

Display options also let you change the color and appearance of different script elements. These include comments, strings, WinRunner reserved words, operators, and numbers. For each script element, you can assign colors, text attributes (bold, italic, underline), font, and font size. For example, you could display all strings in the color red.

Finally, there are display options that let you control how the hard copy of your scripts will appear when printed.

Personalizing Script Editing Commands

WinRunner includes a list of default keyboard commands that let you move the cursor, delete characters, and cut, copy, and paste information to and from the clipboard. You can replace these commands with commands you prefer. For example, you could change the Set Bookmark [#] command from the default CTRL + K + [#] TO CTRL + B + [#].

Setting Display Options

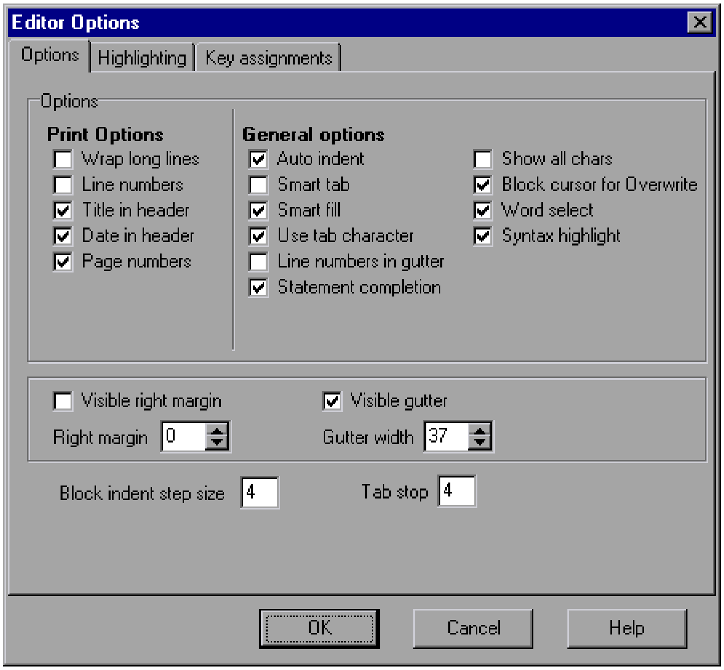
WinRunner's display options let you control how test scripts appear in test windows, how different elements of test scripts are displayed, and how test scripts will appear when they are printed.

Customizing Test Scripts and Windows

You can customize the appearance of WinRunner's test windows and how your scripts are displayed. For example, you can set the size of the test window margins, highlight script elements, and show or hide text symbols.

To customize the appearance of your script:

1 Choose **Tools** > **Editor Options**. The Editor Options dialog box opens.



Options

Print Options
- ☐ Wrap long lines
- ☐ Line numbers
- ☑ Title in header
- ☑ Date in header
- ☑ Page numbers

General options
- ☑ Auto indent
- ☐ Smart tab
- ☑ Smart fill
- ☑ Use tab character
- ☐ Line numbers in gutter
- ☑ Statement completion
- ☐ Show all chars
- ☑ Block cursor for Overwrite
- ☑ Word select
- ☑ Syntax highlight

☐ Visible right margin
Right margin 0

☑ Visible gutter
Gutter width 37

Block indent step size 4

Tab stop 4

2 Click the **Options** tab.

3 Under the **General options** choose from the following options:

Options	**Description**
Auto indent	Causes lines following an indented line to automatically begin at the same point as the previous line. You can click the Home key on your keyboard to move the cursor back to the left margin.
Smart tab	A single press of the tab key will insert the appropriate number of tabs and spaces in order to align the cursor with the text in the line above.

Options	Description
Smart fill	Insert the appropriate number of tabs and spaces in order to apply the Auto indent option. When this option is not selected, only spaces are used to apply the Auto indent. **Note:** Both **Auto indent** and **Use tab character** must be selected to apply this option.
Use tab character	Inserts a tab character when the tab key on the keyboard is used. When this option is not enabled, the appropriate number of space characters will be inserted instead.
Line numbers in gutter	Displays a line number next to each line in the script. The line number is displayed in the test script window's gutter.
Statement completion	Opens a list box displaying all available matches to the function prefix whenever the user presses the CTRL and SPACE keys simultaneously, or presses the Underscore key. Select an item from the list to replace the typed string. To close the list box, press the ESC key. Displays a tooltip with the function parameters once the complete function name appears in the editor.
Show all chars	Displays all text symbols, such as tabs and paragraph symbols.
Block cursor for Overwrite	Displays a block cursor instead of the standard cursor when you select overwrite mode.
Word select	Selects the nearest word when you double-click on the test window.
Syntax highlight	Highlights script elements such as comments, strings, or reserved words. For information on reserved words, see "Reserved Words" on page 257.
Visible right margin	Displays a line that indicates the test window's right margin.

Options	Description
Right margin	Sets the position, in characters, of the test window's right margin (0=left window edge).
Visible gutter	Displays a blank area (gutter) in the test window's left margin.
Gutter width	Sets the width, in pixels, of the gutter.
Block indent step size	Sets the number characters that the selected block of TSL statements will be moved (indented) when the INDENT SELECTED BLOCK softkey is used. For more information on editor softkeys, see "Personalizing Editing Commands" on page 259.
Tab stop	Sets the distance, in characters, between each tab stop.

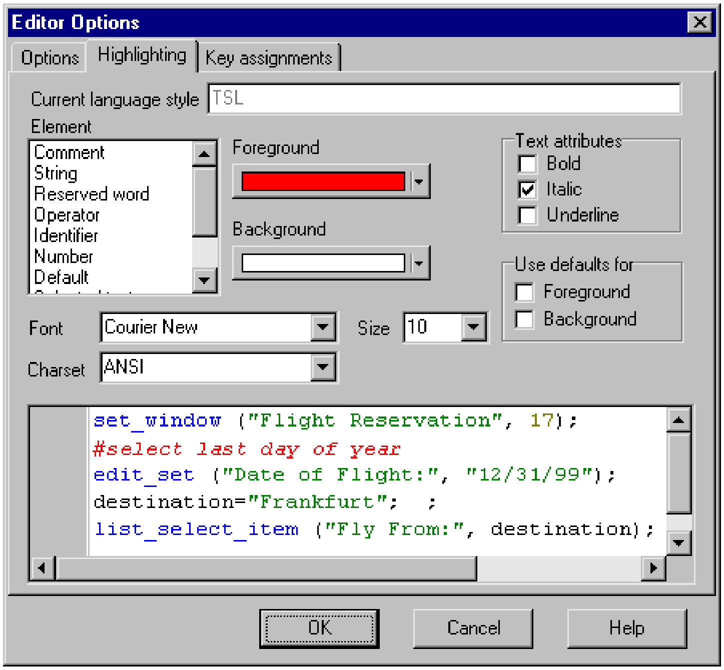
Highlighting Script Elements

WinRunner scripts contain many different elements, such as comments, strings, WinRunner reserved words, operators and numbers. Each element of a WinRunner script is displayed in a different color and style. You can create your own personalized color scheme and style for each script element. For example, all comments in your scripts could be displayed as italicized, blue letters on a yellow background.

To edit script elements:

1 Choose **Tools** > **Editor Options**. The Editor Options dialog box opens.

2 Click the **Highlighting** tab.



Current language style: TSL

Element: Comment, String, Reserved word, Operator, Identifier, Number, Default

Text attributes: ☐ Bold ☑ Italic ☐ Underline

Use defaults for: ☐ Foreground ☐ Background

Font: Courier New

Size: 10

Charset: ANSI

```
set_window ("Flight Reservation", 17);
#select last day of year
edit_set ("Date of Flight:", "12/31/99");
destination="Frankfurt";  ;
list_select_item ("Fly From:", destination);
```

3 Select a script element from the **Element** list.

4 Choose from the following options:

Options	Description
Foreground	Sets the color applied to the text of the script element.
Background	Sets the color that appears behind the script element.
Text Attributes	Sets the text attributes applied to the script element. You can select bold, italic, or underline or a combination of these attributes.
Use defaults for	Applies the font and colors of the “default” style to the selected style.
Font	Sets the typeface of all script elements.

Options	Description
Size	Set the size, in points, of all script elements.
Charset	Sets the character subset of the selected font.

An example of each change you apply will be displayed in the pane at the bottom of the dialog box.

5 Click **OK** to apply the changes.

Reserved Words

WinRunner contains "reserved words," which include the names of all TSL functions and language keywords, such as auto, break, char, close, continue, int, function. For a complete list of all reserved words in WinRunner, refer to the *TSL Reference*. You can add your own reserved words in the *[ct_KEYWORD_USER]* section of the *reserved_words.ini* file, which is located in the *dat* folder in the WinRunner installation directory. Use a text editor, such as Notepad, to open the file. Note that after editing the list, you must restart WinRunner so that it will read from the updated list.

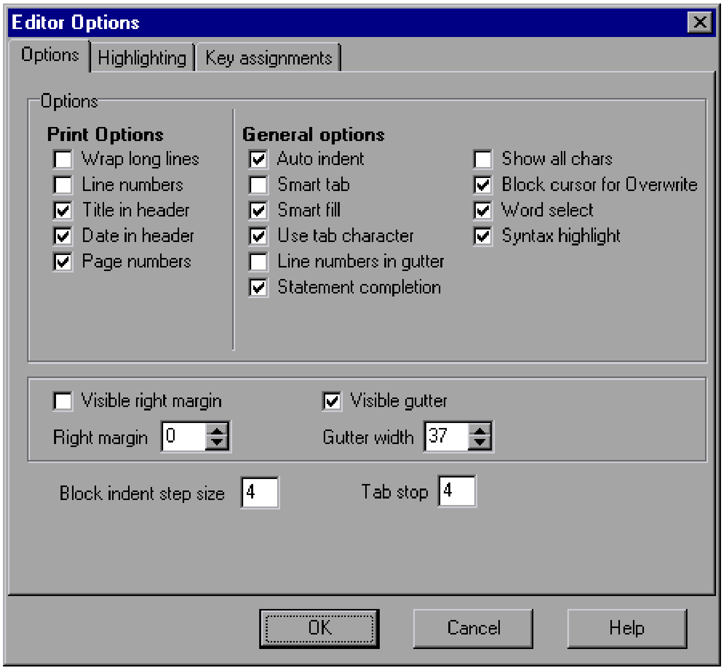
Customizing Print Options

You can set how the hard copy of your script will appear when it is sent to the printer. For example, your printed script can include line numbers, the name of the file, and the date it was printed.

To customize your print options:

1 Choose **Tools** > **Editor Options**. The Editor Options dialog box opens.

2 Click the **Options** tab.



Options

Print Options

☐ Wrap long lines
☐ Line numbers
☑ Title in header
☑ Date in header
☑ Page numbers

General options

☑ Auto indent
☐ Smart tab
☑ Smart fill
☑ Use tab character
☐ Line numbers in gutter
☑ Statement completion
☐ Show all chars
☑ Block cursor for Overwrite
☑ Word select
☑ Syntax highlight

☐ Visible right margin
Right margin 0

☑ Visible gutter
Gutter width 37

Block indent step size 4
Tab stop 4

3 Choose from the following Print options:

Option	Description
Wrap long lines	Automatically wraps a line of text to the next line if it is wider than the current printer page settings.
Line numbers	Prints a line number next to each line in the script.
Title in header	Inserts the file name into the header of the printed script.
Date in header	Inserts today's date into the header of the printed script.
Page numbers	Numbers each page of the script.

4 Click **OK** to apply the changes.

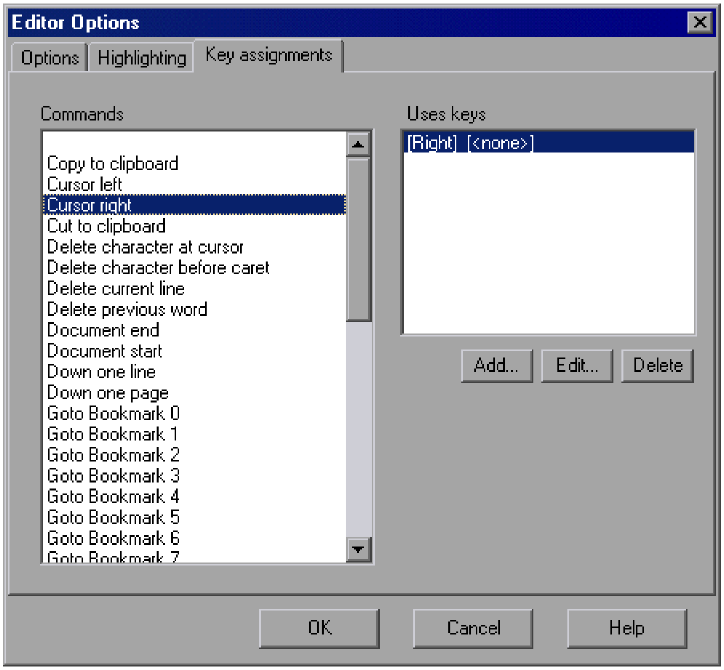
Personalizing Editing Commands

You can personalize the default keyboard commands you use for editing test scripts. WinRunner includes keyboard commands that let you move the cursor, delete characters, and cut, copy, and paste information to and from the clipboard. You can replace these commands with your own preferred commands. For example, you could change the Paste command from the default Ctrl + V to Ctrl + P.

To personalize editing commands:

1 Choose **Tools** > **Editor Options**. The Editor Options dialog box opens.

2 Click the **Key assignments** tab.

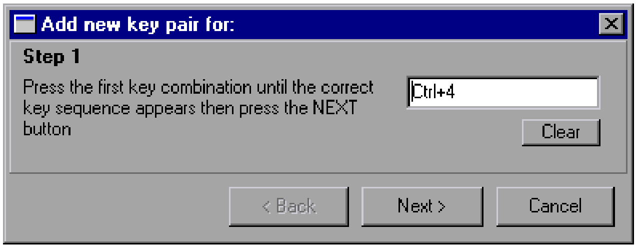


Copy to clipboard
Cursor left
Cursor right
Cut to clipboard
Delete character at cursor
Delete character before caret
Delete current line
Delete previous word
Document end
Document start
Down one line
Down one page
Goto Bookmark 0
Goto Bookmark 1
Goto Bookmark 2
Goto Bookmark 3
Goto Bookmark 4
Goto Bookmark 5
Goto Bookmark 6
Goto Bookmark 7

[Right] [<none>]

3 Select a command from the **Commands** list.

4 Click **Add** to create an additional key assignment or click **Edit** to modify the existing assignment. The Add/Edit key pair for dialog box opens. Press the keys you want to use, for example, CTRL + 4:



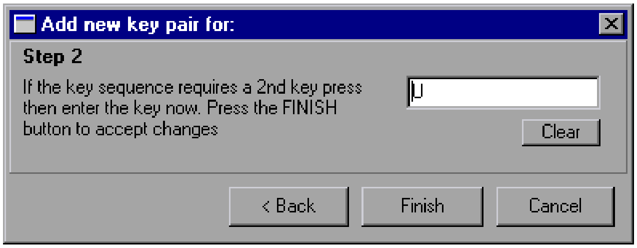
Add new key pair for:

Step 1

Press the first key combination until the correct key sequence appears then press the NEXT button

Ctrl+4

5 Click **Next**. To add an additional key sequence, press the keys you want to use, for example U:



Add new key pair for:

Step 2

If the key sequence requires a 2nd key press then enter the key now. Press the FINISH button to accept changes

U

6 Click **Finish** to add the key sequence(s) to the **Use keys** list.

If you want to delete a key sequence from the list, highlight the keys in the **Uses keys** list and click **Delete**.

7 Click **OK** to apply the changes.

20

Customizing the WinRunner User Interface

You can customize the WinRunner user interface to adapt it to your testing needs and to the application you are testing.

This chapter describes:

- About Customizing WinRunner's User Interface
- Customizing the File, Debug, and User-Defined Toolbars
- Customizing the User Toolbar
- Using the User Toolbar
- Configuring WinRunner Softkeys

About Customizing WinRunner's User Interface

You can adapt WinRunner's user interface to your testing needs by changing the way you access WinRunner commands.

You may find that when you create and run tests, you frequently use the same WinRunner menu commands and insert the same TSL statements into your test scripts. You can create shortcuts to these commands and TSL statements by customizing the WinRunner toolbars.

The application you are testing may use softkeys that are preconfigured for WinRunner commands. If so, you can adapt the WinRunner user interface to this application by using the WinRunner Softkey utility to reconfigure the conflicting WinRunner softkeys.

Customizing the File, Debug, and User-Defined Toolbars

You can use the Customize Toolbars option to create user-defined toolbars and to customize the appearance and contents of the File, Debug, and user-defined toolbars.

Note: You can also customize the User toolbar. For more information, see "Customizing the User Toolbar" on page 269.

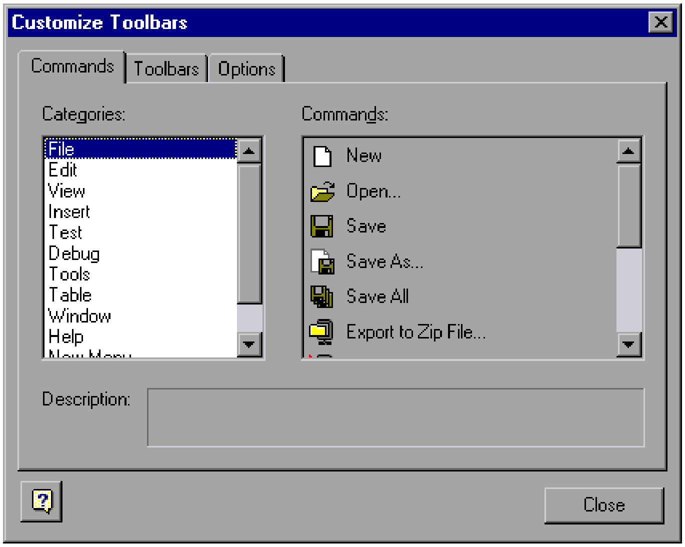
Adding or Removing Toolbar Buttons that Perform Menu Commands

Using the **Commands** tab of the Customize Toolbars dialog box, you can add toolbar buttons that perform frequently-used menu commands to the File and Debug toolbars or to any existing user-defined toolbars. You can also remove toolbar buttons from any of these toolbars.

Tip: You can restore the default buttons to a selected toolbar or to all toolbars using the **Reset** or **Reset All** buttons in the **Toolbars** tab. For more information, see "Controlling the Toolbars Display" on page 264.

To add a button to the File, Debug, or User-Defined Toolbars:

1. Choose **View** > **Customize Toolbars**. The Customize Toolbars dialog box opens and displays the **Commands** tab.



2. In the **Categories** list, find and select the menu name that contains the command you want to add to the toolbar.

3. In the **Commands** list, select the command you want to add and drag it to the File, Debug, or User-Defined toolbar.

4. When you place the button over one of these toolbars, the mouse pointer becomes an I-beam cursor, indicating the location where the button will be placed. Drag the I-beam cursor to the location where you want to add the button, and release the mouse button.

Tip: You can also drag toolbar buttons from one toolbar to another toolbar while the Customize Toolbars dialog box is open.

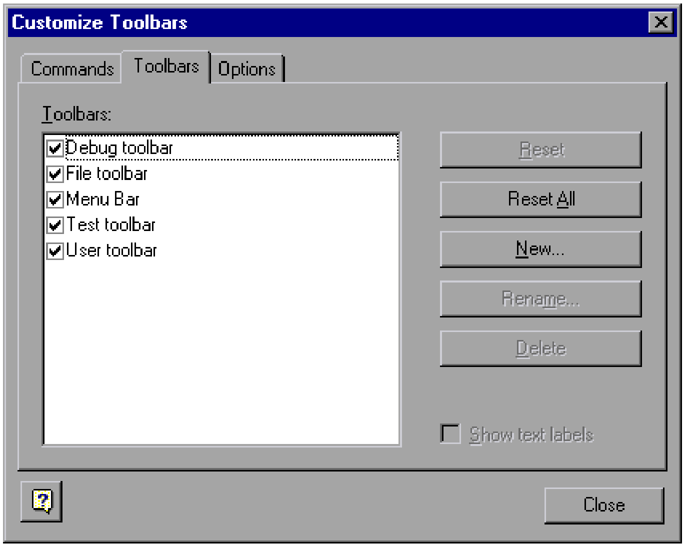
To remove a button from the File, Debug, or User-Defined Toolbars:

1. Choose **View** > **Customize Toolbars**. The Customize Toolbars dialog box opens.
2. Drag the toolbar button you want to remove from the toolbar to any location outside the toolbars area. The toolbar is removed.

Controlling the Toolbars Display

The **Toolbars** tab of the Customize Toolbars dialog box enables you to display or hide toolbars; restore the default buttons on toolbars; create, rename, and delete user-defined toolbars; and control the appearance of individual toolbars.

Tip: You can also display or hide WinRunner toolbars using the appropriate option in the **View** menu.



Toolbars:

☑ Debug toolbar
☑ File toolbar
☑ Menu Bar
☑ Test toolbar
☑ User toolbar

□ Show text labels

To display or hide a toolbar:

1. Choose **View** > **Customize Toolbars**. The Customize Toolbars dialog box opens and displays the **Commands** tab.
2. Click the **Toolbars** tab.
3. Select or clear the check box next to a WinRunner or user-defined toolbar to display or hide it.

Note: You cannot hide the **Menu** bar.

To restore the default buttons on one or all WinRunner toolbars:

1. Choose **View** > **Customize Toolbars**. The Customize Toolbars dialog box opens and displays the **Commands** tab.
2. Click the **Toolbars** tab.
3. To restore the default buttons for a specific toolbar, select the toolbar from the toolbars list and click **Reset**.

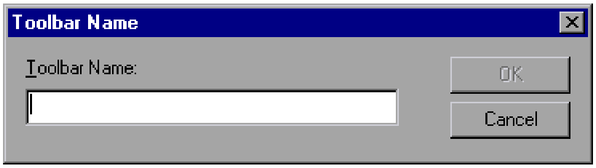
Note: The **Reset** button is disabled if a user-defined toolbar is selected.

To restore the default buttons for all WinRunner toolbars, click **Reset All**.

To create a user-defined toolbar:

1. Choose **View** > **Customize Toolbars**. The Customize Toolbars dialog box opens and displays the **Commands** tab.
2. Click the **Toolbars** tab.

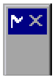
3 Click **New**. The Toolbar Name dialog box opens.



Toolbar Name

Toolbar Name:

4 Enter a unique name for the toolbar and click **OK**. The name of the new toolbar is added to the Toolbars list. The new, blank toolbar opens as a *floating* toolbar in the middle of your screen.



5 Drag the toolbar to the location where you want to keep it. If you drag the toolbar to a location within the top or right-hand toolbar area, it becomes a *docked* toolbar (the titlebar is replaced with a toolbar handle).

Tip: You can also double-click the titlebar to dock the toolbar in a default location in the top toolbar area.

---

6 Use the **Commands** tab of the Customize Toolbars dialog box to add toolbar buttons to your new toolbar. For more information, see “Adding or Removing Toolbar Buttons that Perform Menu Commands” on page 262.

To rename a user-defined toolbar:

1 Choose **View** > **Customize Toolbars**. The Customize Toolbars dialog box opens and displays the **Commands** tab.

2 Click the **Toolbars** tab.

3 Select the user-defined toolbar you want to rename.

Note: The **Rename** option is enabled only when a user-defined toolbar is selected.

---

4 Click **Rename**. The Toolbar Name dialog box opens and displays the current name of the selected toolbar.

5 Enter a new name and click **OK**.

To delete a user-defined toolbar:

1 Choose **View** > **Customize Toolbars**. The Customize Toolbars dialog box opens and displays the **Commands** tab.

2 Click the **Toolbars** tab.

3 Select the user-defined toolbar you want to rename.

Note: The **Delete** option is enabled only when a user-defined toolbar is selected.

---

4 Click **Delete**.

5 Click **Yes** to confirm that you want to delete the selected toolbar. The toolbar is deleted from the toolbars list and from the WinRunner window.

To display text labels on the Debug, File, or Test Toolbars:

1 Choose **View** > **Customize Toolbars**. The Customize Toolbars dialog box opens and displays the **Commands** tab.

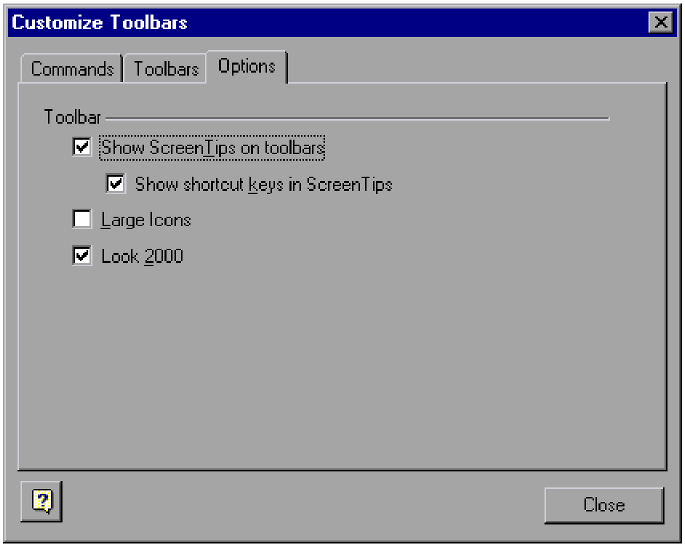
2 Click the **Toolbars** tab.

3 Select the **Debug**, **File**, or **Test** toolbar from the Toolbars list.

4 Select the **Show text labels** check box.

Setting Toolbar Options

The **Options** tab of the Customize Toolbars dialog box enables you to set options that apply to all toolbars.



Toolbar

☑ Show ScreenTips on toolbars

☑ Show shortcut keys in ScreenTips

☐ Large Icons

☑ Look 2000

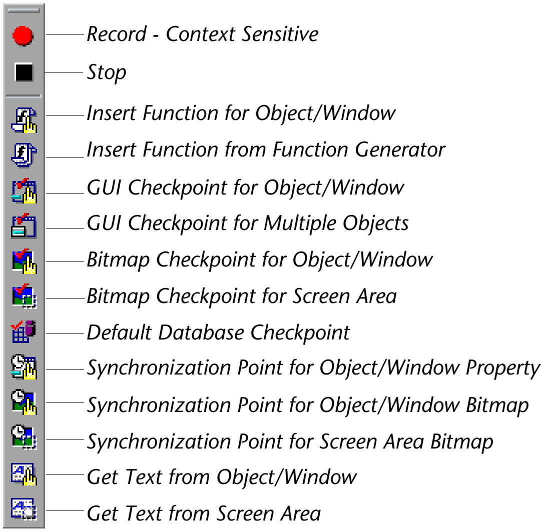
The **Options** tab contains the following options:

Option	Description
Show ScreenTips on toolbars	Shows tips containing the name of the command represented by a toolbar button when you point to the button with the mouse.
Show shortcut keys in ScreenTips	Shows the shortcut key for the command represented by a toolbar button in its screen tip. Enabled only when **Show ScreenTips on toolbars** is selected

Option	Description
Large Icons	Displays all toolbar buttons using large icons.
Look 2000	When selected, displays toolbar handles in the Windows 2000 style with one bar. When cleared, displays toolbar handles with two bars. This option is available only when the **Default** theme is selected in the **Appearance** category of the General Options dialog box.

Customizing the User Toolbar

The User toolbar contains buttons for commands used when creating tests. In its default setting, the User toolbar enables easy access to the following WinRunner commands:


Record - Context Sensitive
Stop
Insert Function for Object/Window
Insert Function from Function Generator
GUI Checkpoint for Object/Window
GUI Checkpoint for Multiple Objects
Bitmap Checkpoint for Object/Window
Bitmap Checkpoint for Screen Area
Default Database Checkpoint
Synchronization Point for Object/Window Property
Synchronization Point for Object/Window Bitmap
Synchronization Point for Screen Area Bitmap
Get Text from Object/Window
Get Text from Screen Area

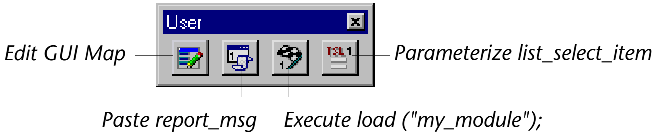
By default, the User toolbar is hidden. To display the User toolbar, choose **View** > **User Toolbar** or select **User Toolbar** in the **Toolbars** tab of the Customize Toolbars dialog box (**View** > **Customize Toolbars**). When the User toolbar is displayed, its default position is docked at the right edge of the WinRunner window.

The User toolbar is a customizable toolbar. You can add or remove buttons to facilitate access to the commands you most frequently use when testing an application. You can use the User toolbar to:

- execute additional WinRunner menu commands. For example, you can add a button to the User toolbar that opens the GUI Map Editor.
- paste TSL statements into your test scripts. For example, you can add a button to the User toolbar that pastes the TSL statement **report_msg** into your test scripts.
- execute TSL statements. For example, you can add a button to the User toolbar that executes the TSL statement:

  ```
  load ("my_module");
  ```

- parameterize TSL statements before pasting them into your test scripts or executing them. For example, you can add a button to the User toolbar that enables you to add parameters to the TSL statement **list_select_item**, and then either paste it into your test script or execute it.



Note: None of the buttons that appear by default in the User toolbar appear in the illustration above.

Adding Buttons to the User Toolbar that Perform Menu Commands

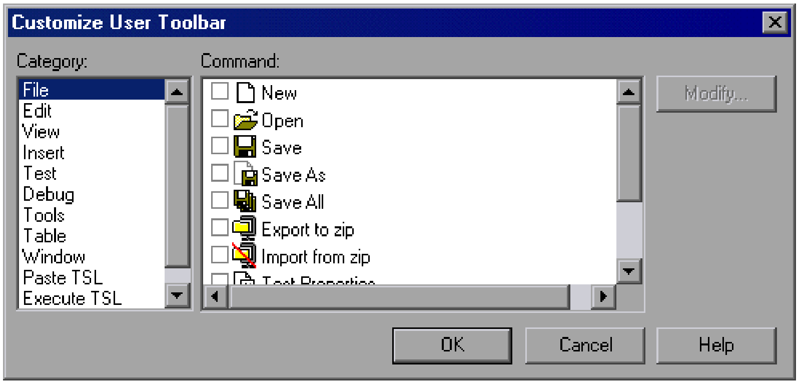
You can add buttons to the User toolbar that perform frequently-used menu commands using the Customize User Toolbar dialog box.

Note: You can also add buttons to the **File** and **Debug** toolbars, and you can create user-defined toolbars. For more information, see “Customizing the File, Debug, and User-Defined Toolbars” on page 262.

To add a menu command to the User toolbar:

1 Choose **View** > **Customize User Toolbar**.

The Customize User Toolbar dialog box opens.



Note that each menu in the menu bar corresponds to a category in the **Category** pane of the Customize User Toolbar dialog box.

2 In the **Category** pane, select a menu.

3 In the **Command** pane, select the check box next to the menu command.

4 Click **OK** to close the Customize User Toolbar dialog box.

The selected menu command button is added to the User toolbar.

To remove a menu command from the User toolbar:

1 Choose **View** > **Customize User Toolbar** to open the Customize User Toolbar dialog box.

2 In the **Category** pane, select a menu.

3 In the **Command** pane, clear the check box next to the menu command.

4 Click **OK** to close the Customize User Toolbar dialog box.

The selected menu command button is removed from the User toolbar.

Tip: You can also restore the default buttons to the User toolbar using the **Reset** or **Reset All** buttons in the **Toolbars** tab of the Customize Toolbars dialog box. For more information, see "Controlling the Toolbars Display" on page 264.

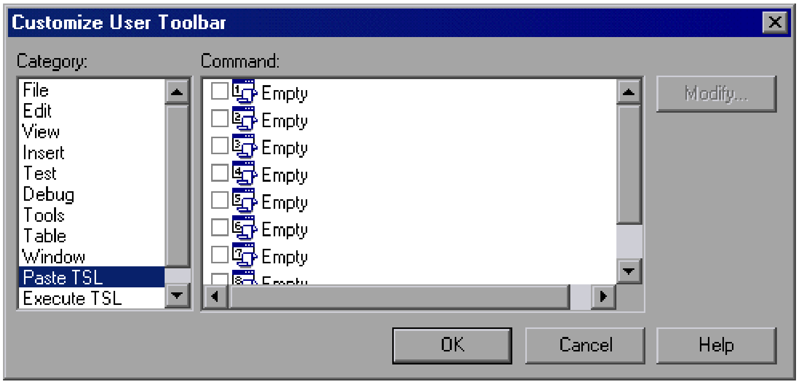
Adding Buttons that Paste TSL Statements

You can add buttons to the User toolbar that paste TSL statements into test scripts. One button can paste a single TSL statement or a group of statements.

To add a button to the User toolbar that pastes TSL statements:

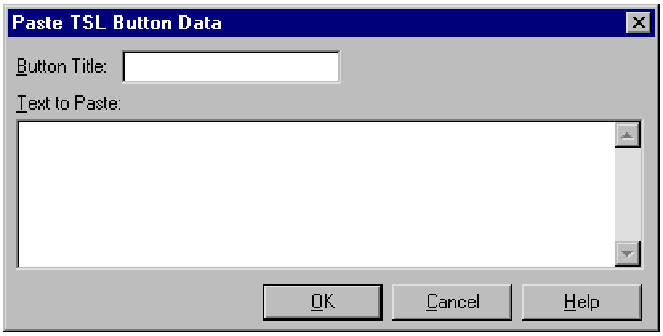
1 Choose **View** > **Customize User Toolbar**. The Customize User Toolbar dialog box opens.

2 In the **Category** pane, select **Paste TSL**.



3 In the **Command** pane, select the check box next to a button, and then select the button.

4 Click **Modify.** The Paste TSL Button Data dialog box opens.



5 In the **Button Title** box, enter a name for the button.

6 In the **Text to Paste** pane, enter the TSL statement(s).

7 Click **OK** to close the Paste TSL Button Data dialog box.

The name of the button is displayed beside the corresponding button in the Command pane.

8 Click **OK** to close the Customize User Toolbar dialog box. The button is added to the User toolbar.

To modify a button on the User toolbar that pastes TSL statements:

1. Choose **View** > **Customize User Toolbar** to open the Customize User Toolbar dialog box.
2. In the **Category** pane, select **Paste TSL**.
3. In the **Command** pane, select the button whose content you want to modify.
4. Click **Modify**. The Paste TSL Button Data dialog box opens.
5. Enter the desired changes in the **Button Title** box and/or the **Text to Paste** pane.
6. Click **OK** to close the Paste TSL Button Data dialog box.
7. Click **OK** to close the Customize User Toolbar dialog box. The button on the User toolbar is modified.

To remove a button from the User toolbar that pastes TSL statements:

1. Choose **View** > **Customize User Toolbar** to open the Customize User Toolbar dialog box.
2. In the **Category** pane, select **Paste TSL**.
3. In the **Command** pane, clear the check box next to the button.
4. Click **OK** to close the Customize User Toolbar dialog box. The button is removed from the User toolbar.

Adding Buttons that Execute TSL Statements

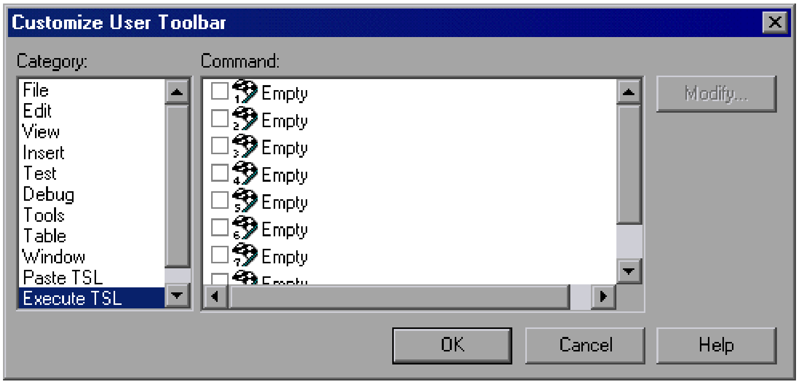
You can add buttons to the User toolbar that execute frequently-used TSL statements.

To add a button to the User toolbar that executes a TSL statement:

1. Choose **View** > **Customize User Toolbar**.

   The Customize User Toolbar dialog box opens.

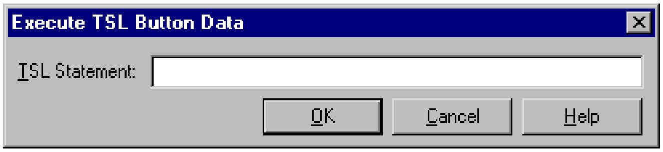
2 In the **Category** pane, select **Execute TSL**.



3 In the **Command** pane, select the check box next to a button, and then select the button.

4 Click **Modify**.

The Execute TSL Button Data dialog box opens.



5 In the **TSL Statement** box, enter the TSL statement.

6 Click **OK** to close the Execute TSL Button Data dialog box.

The TSL statement is displayed beside the corresponding button in the Command pane.

7 Click **OK** to close the Customize User Toolbar dialog box. The button is added to the User toolbar.

To modify a button on the User toolbar that executes a TSL statement:

1 Choose **View** > **Customize User Toolbar** to open the Customize User Toolbar dialog box.

2 In the **Category** pane, select **Execute TSL**.

3. In the **Command** pane, select the button whose content you want to modify.
4. Click **Modify**. The Execute TSL Button Data dialog box opens.
5. Enter the desired changes in the **TSL Statement** box.
6. Click **OK** to close the Execute TSL Button Data dialog box.
7. Click **OK** to close the Customize User Toolbar dialog box. The button on the User toolbar is modified.

To remove a button from the User toolbar that executes a TSL statement:

1. Choose **View** > **Customize User Toolbar** to open the Customize User Toolbar dialog box.
2. In the **Category** pane, select **Execute TSL**.
3. In the **Command** pane, clear the check box next to the button.
4. Click **OK** to close the Customize User Toolbar dialog box. The button is removed from the User toolbar.

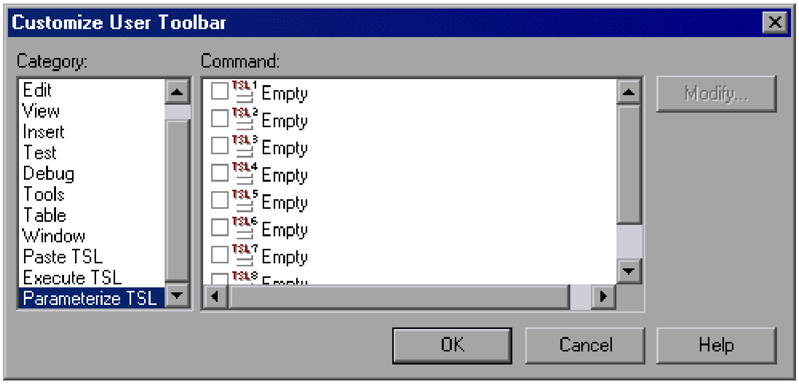
Adding Buttons that Parameterize TSL Statements

You can add buttons to the User toolbar that enable you to easily parameterize frequently-used TSL statements, and then paste them into your test script or execute them.

To add a button to the User toolbar that enables you to parameterize a TSL statement:

1. Choose **View** > **Customize User Toolbar**. The Customize User Toolbar dialog box opens.

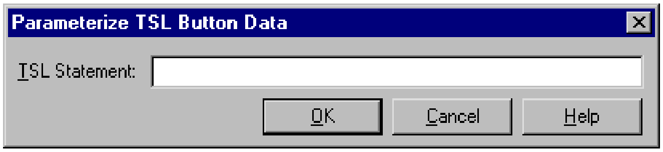
2 In the **Category** pane, select **Parameterize TSL**.



3 In the **Command** pane, select the check box next to a button, and then select the button.

4 Click **Modify.**

The Parameterize TSL Button Data dialog box opens.



5 In the **TSL Statement** box, enter the name of TSL function. You do not need to enter any parameters. For example, enter **list_select_item**.

6 Click **OK** to close the Parameterize TSL Button Data dialog box. The TSL statement is displayed beside the corresponding button in the Command pane.

7 Click **OK** to close the Customize User Toolbar dialog box. The button is added to the User toolbar.

To modify a button on the User toolbar that enables you to parameterize a TSL statement:

1 Choose **View** > **Customize User Toolbar** to open the Customize User Toolbar dialog box.

2 In the **Category** pane, select **Parameterize TSL**.

3 In the **Command** pane, select the button whose content you want to modify.

4 Click **Modify**. The Parameterize TSL Button Data dialog box opens.

5 Enter the desired changes in the **TSL Statement** box.

6 Click **OK** to close the Parameterize TSL Button Data dialog box.

7 Click **OK** to close the Customize User Toolbar dialog box. The button on the User toolbar is modified.

To remove a button from the User toolbar that enables you to parameterize a TSL statement:

1 Choose **View** > **Customize User Toolbar** to open the Customize User Toolbar dialog box.

2 In the **Category** pane, select **Parameterize TSL**.

3 In the **Command** pane, clear the check box next to the button.

4 Click **OK** to close the Customize User Toolbar dialog box. The button is removed from the User toolbar.

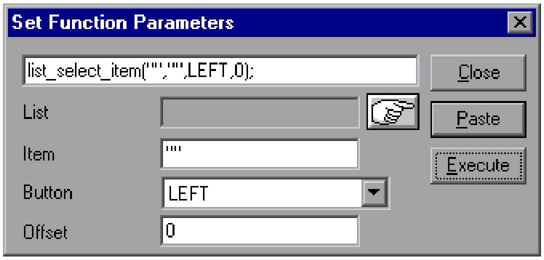
Using the User Toolbar

The User toolbar is hidden by default. You can display it by selecting it from the **View** menu. To execute a command on the User toolbar, click the button that corresponds to the command you want. You can also access the same TSL-based commands that appear on the User toolbar by choosing them on the **Insert** menu.

When the User toolbar is a "floating" toolbar, it remains open when you minimize WinRunner while recording a test. For more information, refer to Chapter 8, "Designing Tests" in the *Mercury WinRunner Basic Features User's Guide*.

Parameterizing a TSL Statement

When you click a button on the User toolbar that represents a TSL statement to be parameterized, the Set Function Parameters dialog box opens.



Set Function Parameters

list_select_item("","",LEFT,0);

List

Item ""

Button LEFT

Offset 0

The Set Function Parameters dialog box varies in its appearance according to the parameters required by a particular TSL function. For example, the **list_select_item** function has four parameters: *List*, *Item*, *Button* and *Offset*. For each parameter, you define a value as described below:

- To define a value for the *List* parameter, click the pointing hand. WinRunner is minimized, a help window opens, and the mouse pointer becomes a pointing hand. Click the list in your application.
- To define a value for the *Item* parameter, type it in the corresponding box.
- To define a value for the *Button* parameter, select it from the list.
- To define a value for the *Offset* parameter, type it in the corresponding box.

Accessing TSL Statements on the Menu Bar

All TSL statements that you add to the User toolbar can also be accessed via the **Insert** menu.

To choose a TSL statement from a menu:

- To paste a TSL statement, choose **Insert** > **Paste TSL** > [TSL Statement].
- To execute a TSL statement, choose **Insert** > **Execute TSL** > [TSL Statement].
- To parameterize a TSL statement, choose **Insert** > **Parameterize TSL** > [TSL Statement].

Configuring WinRunner Softkeys

Several WinRunner commands can be carried out using softkeys. WinRunner can carry out softkey commands even when the WinRunner window is not the active window on your screen, or when it is minimized.

If the application you are testing uses a softkey combination that is preconfigured for WinRunner, you can redefine the WinRunner softkey combination using WinRunner's Softkey Configuration utility.

Default Settings for WinRunner Softkeys

The following table lists the default softkey configurations and their functions.

Command	Default Softkey Combination	Function
RECORD	F2	Starts test recording. While recording, this softkey toggles between Context Sensitive and Analog modes.
CHECK GUI FOR SINGLE PROPERTY	Alt Right + F12	Checks a single property of a GUI object.
CHECK GUI FOR OBJECT/WINDOW	Ctrl Right + F12	Creates a GUI checkpoint for an object or a window.

Command	Default Softkey Combination	Function
CHECK GUI FOR MULTIPLE OBJECTS	F12	Opens the Create GUI Checkpoint dialog box.
CHECK BITMAP OF OBJECT/WINDOW	Ctrl Left + F12	Captures an object or a window bitmap.
CHECK BITMAP OF SCREEN AREA	Alt Left + F12	Captures an area bitmap.
CHECK DATABASE (DEFAULT)	Ctrl Right + F9	Creates a check on the entire contents of a database.
CHECK DATABASE (CUSTOM)	Alt Right + F9	Checks the number of columns, rows and specified information of a database.
SYNCHRONIZE OBJECT/WINDOW PROPERTY	Ctrl Right + F10	Instructs WinRunner to wait for a property of an object or a window to have an expected value.
SYNCHRONIZE BITMAP OF OBJECT/WINDOW	Ctrl Left + F11	Instructs WinRunner to wait for a specific object or window bitmap to appear.
SYNCHRONIZE BITMAP OF SCREEN AREA	Alt Left + F11	Instructs WinRunner to wait for a specific area bitmap to appear.
GET TEXT FROM OBJECT/WINDOW	F11	Captures text in an object or a window.
GET TEXT FROM SCREEN AREA	Alt Right + F11	Captures text in a specified area and adds a **get_text** statement to the test script.
INSERT FUNCTION FOR OBJECT/WINDOW	F8	Inserts a TSL function for a GUI object.
INSERT FUNCTION FROM FUNCTION GENERATOR	F7	Opens the Function Generator dialog box.
RUN FROM TOP	Ctrl Left + F5	Runs the test from the beginning.

Command	Default Softkey Combination	Function
RUN FROM ARROW	Ctrl Left + F7	Runs the test from the line in the script indicated by the arrow.
STEP	F6	Runs only the current line of the test script.
STEP INTO	Ctrl Left + F8	Like Step: however, if the current line calls a test or function, the called test or function is displayed in the WinRunner window but is not executed.
STEP TO CURSOR	Ctrl Left + F9	Runs a test from the line indicated by the arrow to the line marked by the insertion point.
PAUSE	PAUSE	Stops the test run after all previously interpreted TSL statements have been executed. Execution can be resumed from this point using the Run from Arrow command or the RUN FROM ARROW softkey.
STOP	Ctrl Left + F3	Stops test recording or the test run.
MOVE LOCATOR	Alt Left + F6	Records a move_locator_abs statement with the current position (in pixels) of the screen pointer.

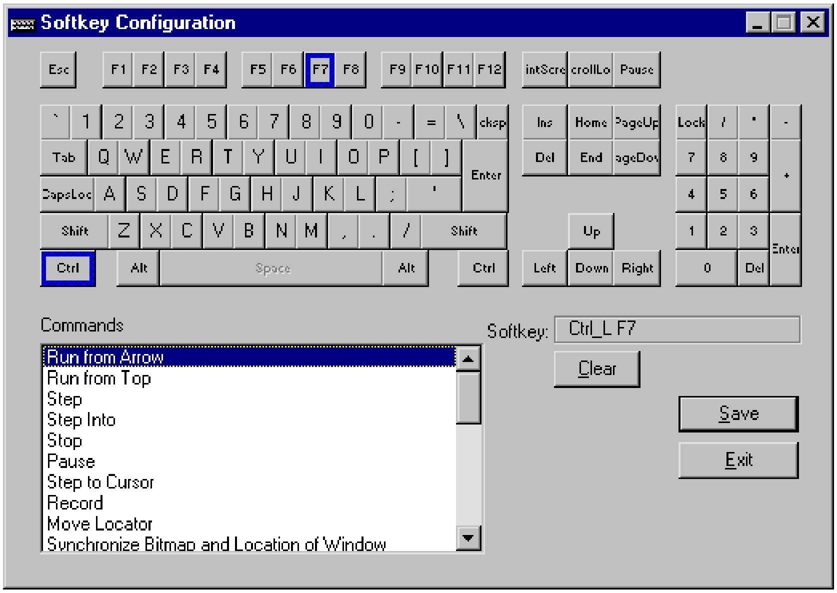
Redefining WinRunner Softkeys

The Softkey Configuration dialog box lists the current softkey assignments and displays an image of a keyboard. To change a softkey setting, click the new key combination as it appears in the dialog box.

To change a WinRunner softkey setting:

1 Choose **Start** > **Programs** > **WinRunner** > **Softkey Configuration**. The Softkey Configuration dialog box opens.

The Commands pane lists all the WinRunner softkey commands.



Commands

- Run from Arrow
- Run from Top
- Step
- Step Into
- Stop
- Pause
- Step to Cursor
- Record
- Move Locator
- Synchronize Bitmap and Location of Window

Softkey: Ctrl_L F7

2 Click the command you want to change. The current softkey definition appears in the **Softkey** box; its keys are highlighted on the keyboard.

3 Click the new key or combination that you want to define. The new definition appears in the **Softkey** box.

An error message appears if you choose a definition that is already in use or an illegal key combination. Click a different key or combination.

4 Click **Save** to save the changes and close the dialog box. The new softkey configuration takes effect when you start WinRunner.

21

Setting Testing Options from a Test Script

You can control how WinRunner records and runs tests by setting and retrieving testing options from within a test script.

This chapter describes:

- About Setting Testing Options from a Test Script
- Setting Testing Options with setvar
- Retrieving Testing Options with getvar
- Controlling the Test Run with setvar and getvar
- Using Test Script Testing Options

About Setting Testing Options from a Test Script

WinRunner testing options affect how you record test scripts and run tests. For example, you can set the speed at which WinRunner executes a test or determine how WinRunner records keyboard input.

You can set and retrieve the values of testing options from within a test script. To set the value of a testing option, use the **setvar** function. To retrieve the current value of a testing option, use the **getvar** function. By using a combination of **setvar** and **getvar** statements in a test script, you can control how WinRunner executes a test. You can use these functions to set and view the testing options for all tests, for a single test, or for part of a single test. You can also use these functions in a startup test script to set environment variables.

Most testing options can also be set using the General Options dialog box. For more information on setting testing options using the General Options dialog box, refer to Chapter 23, "Setting Global Testing Options" in the *Mercury WinRunner Basic Features User's Guide*.

Setting Testing Options with setvar

You use the **setvar** function to set the value of a testing option from within the test script. This function has the following syntax:

setvar ("*testing_option***",** **"***value***");**

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

attached_text_area
attached_text_search_radius
beep
capture_bitmap
cs_run_delay
cs_fail
delay_msec
drop_sync_timeout
email_service
enum_descendent_toplevel
fontgrp
item_number_seq
List_item_separator
Listview_item_separator
min_diff
mismatch_break
rec_item_name
rec_owner_drawn
searchpath
silent_mode
single_prop_check_fail
speed
sync_fail_beep
synchronization_timeout
tempdir
timeout_msec
Treeview_path_separator

For example, if you execute the following **setvar** statement:

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

WinRunner disables the *mismatch_break* testing option. The setting remains in effect during the testing session until it is changed again, either with another **setvar** statement or from the corresponding **Break when verification fails** check box in the **Run** > **Settings** category of the General Options dialog box.

Using the **setvar** function changes a testing option globally, and this change is reflected in the General Options dialog box. However, you can also use the **setvar** function to set testing options for a specific test, or even for part of a specific test.

To use the **setvar** function to change a variable only for the current test, without overwriting its global value, save the original value of the variable separately and restore it later in the test.

For example, if you want to change the *delay_msec* testing option to 20,000 for a specific test only, insert the following at the beginning of your test script:

```
# Keep the original value of the 'delay_msec' testing option
old_delay = getvar ("delay_msec") ;
setvar ("delay_msec", "20,000") ;
```

To change back the *delay* testing option to its original value at the end of the test, insert the following at the end of your test script:

```
#Change back the ‘delay_msec’ testing option to its original value.
setvar (“delay_msec”, old_delay) ;
```

Note: Some testing options are set by WinRunner and cannot be changed through either **setvar** or the General Options dialog box. For example, the value of the testname option is always the name of the current test. You can use **getvar** to retrieve this read-only value. For more information, see “Retrieving Testing Options with getvar” on page 288.

Retrieving Testing Options with getvar

You use the **getvar** function to retrieve the current value of a testing option. The **getvar** function is a read-only function, and does not enable you to alter the value of the retrieved testing option. (To change the value of a testing option in a test script, use the **setvar** function, described above.) The syntax of this statement is:

user_variable = **getvar ("***testing_option***");**

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

attached_text_area
attached_text_search_radius
batch
beep
capture_bitmap
cs_fail
cs_run_delay
curr_dir
delay_msec
drop_sync_timeout
email_service
enum_descendent_toplevel
exp
fontgrp
item_number_seq
key_editing
line_no
List_item_separator
Listview_item_separator
min_diff
mismatch_break
rec_item_name
rec_owner_drawn
result
runmode
searchpath
shared_checklist_dir
single_prop_check_fail
silent_mode
speed
sync_fail_beep
synchronization_timeout
qc_connection
qc_cycle_name
qc_database_name
qc_log_dirname
qc_log_dirname
qc_server_name
qc_test_instance
qc_test_run_id
qc_user_name
tempdir
testname
timeout_msec
Treeview_path_separator

For example:

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

assigns the current value of the run speed to the user-defined variable currspeed.

Controlling the Test Run with setvar and getvar

You can use **getvar** and **setvar** together to control a test run without changing global settings. In the following test script fragment, WinRunner checks the bitmap Img1. The **getvar** function retrieves the values of the *timeout_msec* and *delay_msec* testing options, and **setvar** assigns their values for this **win_check_bitmap** statement. After the window is checked, **setvar** restores the values of the testing options.

```
t = getvar ("timeout_msec");
d = getvar ("delay_msec");
setvar ("timeout_msec", 30000);
setvar ("delay_msec", 3000);
win_check_bitmap ("calculator", Img1, 2, 261,269,93,42);
setvar ("timeout_msec", t);
setvar ("delay_msec", d);
```

Note: You can use the **setvar** and **getvar** functions in a startup test script to set environment variables for a specific WinRunner session. For more information, see Chapter 23, “Initializing Special Configurations.”

Using Test Script Testing Options

This section describes the WinRunner testing options that can be used with the **setvar** and **getvar** functions from within a test script. If you can also use set or view the corresponding option from a dialog box, it is indicated below.

attached_text_area

This option specifies the location on a GUI object from which WinRunner searches for its attached text.

Possible values:

Value	Location on the GUI Object
Default	Top-left corner of regular (English-style) windows; Top-right corner of windows with RTL-style (WS_EX_BIDI_CAPTION) windows.
Top-Left	Top-left corner.
Top	Midpoint of two top corners.
Top-Right	Top-right corner.
Right	Midpoint of two right corners.
Bottom-Right	Bottom-right corner.
Bottom	Midpoint of two bottom corners.
Bottom-Left	Bottom-left corner.
Left	Midpoint of two left corners.

Note: All of the above possible values are text strings.

You can use this option with the **setvar** and **getvar** functions.

You can also set this option using the **Attached Text - Preferred search area** box in the **Record** category of the General Options dialog box as described in Chapter 23, “Setting Global Testing Options” in the *Mercury WinRunner Basic Features User’s Guide*.

Notes: When you run a test, you must use the same values for the attached text options that you used when you recorded the test. Otherwise, WinRunner may not identify the GUI object.

In previous versions of WinRunner, you could not set the preferred search area: WinRunner searched for attached text based on what is now the Default setting for the preferred search area. If backward compatibility is important, choose the Default setting.

attached_text_search_radius

This option specifies the radius from the specified location on a GUI object that WinRunner searches for the static text object that is its attached text.

Possible values: 3 - 300 (pixels)

You can use this option with the **setvar** and **getvar** functions.

You can also set this option using the **Attached Text - Search radius** box in the **Record** category of the General Options dialog box as described in Chapter 23, "Setting Global Testing Options" in the *Mercury WinRunner Basic Features User's Guide*.

Note: When you run a test, you must use the same values for the attached text options that you used when you recorded the test. Otherwise, WinRunner may not identify the GUI object.

batch

This option displays whether WinRunner is running in batch mode. In batch mode, WinRunner suppresses messages during a test run so that a test can run unattended. WinRunner also saves all the expected and actual results of a test run in batch mode in one folder, and displays them in one Test Results window. For more information on the batch testing option, see Chapter 14, “Running Batch Tests.”

For example, if a **set_window** statement is missing from a test script, WinRunner cannot find the specified window. If this option is on and the test is run in batch mode, WinRunner reports an error in the Test Results window and proceeds to run the next statement in the test script. If this option is off and the test is not run in batch mode, WinRunner pauses the test and opens the Run wizard to enable the user to locate the window.

You can use this option with the **getvar** function.

Possible values: on, off (text strings)

You can also set this option using the **Run in batch mode** check box in the **Run** category of the General Options dialog box as described in Chapter 23, “Setting Global Testing Options” in the *Mercury WinRunner Basic Features User’s Guide*.

Note that you can also set this option using the corresponding *-batch* command line option, described in Chapter 15, “Running Tests from the Command Line.”

Note: When you run tests in batch mode, you automatically run them in silent mode. For information about the *silent_mode* testing option, see page 304.

beep

This option determines whether WinRunner beeps when checking any window during a test run.

You can use this option with the **setvar** and **getvar** functions.

Possible values: on, off (text strings)

You can also set this option using the corresponding **Beep when checking a window** check box in the **Run** > **Settings** category of the General Options dialog box as described in Chapter 23, "Setting Global Testing Options" in the *Mercury WinRunner Basic Features User's Guide*.

Note that you can also set this option using the corresponding *-beep* command line option, described in Chapter 15, "Running Tests from the Command Line."

capture_bitmap

This option determines whether WinRunner captures a bitmap whenever a checkpoint fails. When this option is on, WinRunner uses the settings from the **Run** > **Settings** category of the General Options dialog box to determine the captured area for the bitmaps.

You can use this option with the **setvar** and **getvar** functions.

Possible values: on, off (text strings)

You can also set this option using the **Capture bitmap on verification failure** check box in the **Run** > **Settings** category of the General Options dialog box, as described in Chapter 23, "Setting Global Testing Options" in the *Mercury WinRunner Basic Features User's Guide*.

Note that you can also set this option using the corresponding *-capture_bitmap* command line option, described in Chapter 15, "Running Tests from the Command Line."

cs_fail

This option determines whether WinRunner fails a test when Context Sensitive errors occur. A Context Sensitive error is the failure of a Context Sensitive statement during a test. Context Sensitive errors are often due to WinRunner's failure to identify a GUI object.

For example, a Context Sensitive error will occur if you run a test containing a **set_window** statement with the name of a non-existent window. Context Sensitive errors can also occur when window names are ambiguous. For information about Context Sensitive functions, refer to the *TSL Reference*.

You can use this option with the **setvar** and **getvar** functions.

Possible values: 1,0

You can also set this option using the corresponding **Fail test when Context Sensitive errors occur** check box in the **Run** > **Settings** category of the General Options dialog box as described in Chapter 23, "Setting Global Testing Options" in the *Mercury WinRunner Basic Features User's Guide*.

You can also set this option using the corresponding *-cs_fail* command line option, described in Chapter 15, "Running Tests from the Command Line."

cs_run_delay

This option sets the time (in milliseconds) that WinRunner waits between executing Context Sensitive statements when running a test.

You can use this option with the **setvar** and **getvar** functions.

Possible values: numbers 0 and higher

You can also set this option using the corresponding **Delay between execution of CS statements** box in the **Run** > **Synchronization** category of the General Options dialog box as described in Chapter 23, "Setting Global Testing Options" in the *Mercury WinRunner Basic Features User's Guide*.

Note that you can also set this option using the corresponding *-cs_run_delay* command line option, described in Chapter 15, "Running Tests from the Command Line."

curr_dir

This option displays the current working folder for the test.

You can use this option with the **getvar** function.

You can also view the location of the current working folder for the test from the corresponding **Current folder** box in the **Current Test** tab of the Test Properties dialog box, described in Chapter 22, "Setting Properties for a Single Test" in the *Mercury WinRunner Basic Features User's Guide*.

delay_msec

This option sets the sampling interval (in seconds) used to determine that a window is stable before capturing it for a Context Sensitive checkpoint or synchronization point. To be declared stable, a window must not change between two consecutive samplings. This sampling continues until the window is stable or the timeout (as set with the *timeout_msec* testing option) is reached. (Formerly *delay*, which was measured in seconds.)

For example, when the delay is two seconds and the timeout is ten seconds, WinRunner checks the window in the application under test every two seconds until two consecutive checks produce the same results or until ten seconds have elapsed. Setting the value to 0 disables all bitmap checking.

You can use this option with the **setvar** and **getvar** functions.

Possible values: numbers 0 and higher

Note: This option is accurate to within 20-30 milliseconds.

You can also set this option using the corresponding **Delay for window synchronization** option in the **Run** > **Synchronization** category of the General Options dialog box as described in Chapter 23, "Setting Global Testing Options" in the *Mercury WinRunner Basic Features User's Guide*.

Note that you can also set this option using the corresponding *-delay_msec* command line option, described in Chapter 15, "Running Tests from the Command Line."

drop_sync_timeout

determines whether WinRunner minimizes the synchronization timeout (as defined in the **timeout_msec** option) after the first synchronization failure.

Possible values: on, off (text strings)

You can use this option with the **getvar** and **setvar** functions.

You can also set this option using the corresponding **Drop synchronization timeout if failed** check box in the **Run** > **Synchronization** category of the General Options dialog box as described in Chapter 23, "Setting Global Testing Options" in the *Mercury WinRunner Basic Features User's Guide*.

email_service

This option determines whether WinRunner activates the e-mail sending options including the e-mail notifications for checkpoint failures, test failures, and test completed reports as well as any **email_send_msg** statements in the test.

Possible values: on, off (text strings)

You can use this option with the **getvar** and **setvar** functions.

You can also set this option using the corresponding Activate e-mail service check box in the **Notifications** > **E-mail** category of the General Options dialog box as described in Chapter 23, "Setting Global Testing Options" in the *Mercury WinRunner Basic Features User's Guide*.

Note that you can also set this option using the corresponding **-email_service** command line option, described in Chapter 15, "Running Tests from the Command Line."

enum_descendent_toplevel

This option determines whether WinRunner records controls (objects) of a child object whose parent is an object but not a window and identifies these controls when running a test.

Possible values: 1,0

You can use this option with the **getvar** and **setvar** functions.

You can also set this option using the corresponding **Consider child windows** check box in the **Record** category of the General Options dialog box as described in Chapter 23, "Setting Global Testing Options" in the *Mercury WinRunner Basic Features User's Guide*.

exp

This option displays the full path of the expected results folder associated with the current test run.

You can use this option with the **getvar** function.

You can also view the full path of the expected results folder from the corresponding **Expected results folder** box in the **Current Test** tab of the Test Properties dialog box as described in Chapter 23, "Setting Global Testing Options" in the *Mercury WinRunner Basic Features User's Guide*.

Note that you can also set this option using the corresponding *-exp* command line option, described in Chapter 15, "Running Tests from the Command Line."

fontgrp

To be able to use Image Text Recognition (instead of the default Text Recognition), (described in Chapter 23, "Setting Global Testing Options" in the *Mercury WinRunner Basic Features User's Guide*), you must choose an active font group. This option sets the active font group for Image Text Recognition. For more information on font groups, refer to Chapter 16, "Checking Text" in the *Mercury WinRunner Basic Features User's Guide*.

You can use this option with the **setvar** and **getvar** functions.

Possible values: any text string

You can also set this option using the corresponding **Font group** box in the **Record** > **Text Recognition** category of the General Options dialog box as described in Chapter 23, “Setting Global Testing Options” in the *Mercury WinRunner Basic Features User’s Guide*.

Note that you can also set this option using the corresponding *-fontgrp* command line option, described in Chapter 15, “Running Tests from the Command Line.”

item_number_seq

This option defines the string recorded in the test script to indicate that a List, ListView, or TreeView item is specified by its index number.

You can use this option with the **setvar** and **getvar** functions.

Possible values: any text string

You can also set this option using the corresponding **String indicating that what follows is a number** box in the **Record** > **Script Format** category of the General Options dialog box as described in Chapter 23, “Setting Global Testing Options” in the *Mercury WinRunner Basic Features User’s Guide*.

key_editing

This option determines whether WinRunner generates more concise **type**, **win_type**, and **obj_type** statements in a test script.

When this option is on, WinRunner generates more concise **type**, **win_type**, and **obj_type** statements that represent only the net result of pressing and releasing input keys. This makes your test script easier to read.

For example:

```
obj_type (object, "A");
```

When this option is disabled, WinRunner records the pressing and releasing of each key. For example:

```
obj_type (object, "<kShift_L>-a-a+<kShift_L>+");
```

Disable this option if the exact order of keystrokes is important for your test.

For more information on this subject, see the **type** function in the *TSL Reference*.

You can use this option with the **setvar** and **getvar** functions.

Possible values: on, off (text strings)

You can also set this option using the corresponding **Generate concise, more readable type statements** check box in the **Record** > **Script Format** category of the General Options dialog box as described in Chapter 23, "Setting Global Testing Options" in the *Mercury WinRunner Basic Features User's Guide*.

line_no

This option displays the line number of the current location of the execution arrow in the test script.

You can use this option with the **getvar** function.

You can also view the current line number in the test script from the corresponding **Current line number** box in the **Current Test** tab of the Test Properties dialog box, described in Chapter 22, "Setting Properties for a Single Test" in the *Mercury WinRunner Basic Features User's Guide*.

List_item_separator

This option defines the string recorded in the test script to separate items in a list box or a combo box.

You can use this option with the **setvar** and **getvar** functions.

Possible values: any text string

You can also set this option using the corresponding **String for separating ListBox or ComboBox items** box in the **Record** > **Script Format** category of the General Options dialog box as described in Chapter 23, "Setting Global Testing Options" in the *Mercury WinRunner Basic Features User's Guide*.

Listview_item_separator

This option defines the string recorded in the test script to separate items in a ListView or a TreeView.

You can use this option with the **setvar** and **getvar** functions.

Possible values: any text string

You can also set this option using the corresponding **String for separating ListView or TreeView items** box in the **Record** > **Script Format** category of the General Options dialog box as described in Chapter 23, "Setting Global Testing Options" in the *Mercury WinRunner Basic Features User's Guide*.

min_diff

This option defines the number of pixels that constitute the threshold for bitmap mismatch. When this value is set to 0, a single pixel mismatch constitutes a bitmap mismatch.

You can use this option with the **setvar** and **getvar** functions.

Possible values: numbers 0 and higher

You can also set this option using the corresponding **Threshold for difference between bitmaps** box in the **Run** > **Settings** category of the General Options dialog box as described in Chapter 23, "Setting Global Testing Options" in the *Mercury WinRunner Basic Features User's Guide*.

Note that you can also set this option using the corresponding *-min_diff* command line option, described in Chapter 15, "Running Tests from the Command Line."

mismatch_break

This option determines whether WinRunner pauses the test run and displays a message whenever verification fails or whenever any message is generated as a result of a context sensitive statement during a test that is run in **Verify** mode. This option should be used only when working interactively.

For example, if a **set_window** statement is missing from a test script, WinRunner cannot find the specified window. If this option is on, WinRunner pauses the test and opens the Run wizard to enable the user to locate the window. If this option is off, WinRunner reports an error in the Test Results window and proceeds to run the next statement in the test script.

You can use this option with the **setvar** and **getvar** functions.

Possible values: on, off (text strings)

You can also set this option using the corresponding **Break when verification fails** check box in the **Run** > **Settings** category of the General Options dialog box as described in Chapter 23, “Setting Global Testing Options” in the *Mercury WinRunner Basic Features User’s Guide*.

Note that you can also set this option using the corresponding *-mismatch_break* command line option, described in Chapter 15, “Running Tests from the Command Line.”

rec_item_name

This option determines whether WinRunner records non-unique ListBox and ComboBox items by name or by index.

You can use this option with the **setvar** and **getvar** functions.

Possible values: 1,0

You can also set this option using the corresponding **Record non-unique list items by name** check box in the **Record** category of the General Options dialog box as described in Chapter 23, “Setting Global Testing Options” in the *Mercury WinRunner Basic Features User’s Guide*.

Note that you can also set this option using the corresponding *-rec_item_name* command line option, described in Chapter 15, “Running Tests from the Command Line.”

rec_owner_drawn

Since WinRunner cannot identify the class of owner-drawn buttons, it automatically maps them to the general "object" class. This option enables you to map all owner-drawn buttons to a standard button class (push_button, radio_button, or check_button).

You can use this option with the **setvar** and **getvar** functions.

Possible Values: object, push_button, radio_button, check_button (text strings)

You can also set this option using the corresponding **Record owner-drawn buttons as** box in the **Record** category of the General Options dialog box as described in Chapter 23, "Setting Global Testing Options" in the *Mercury WinRunner Basic Features User's Guide*.

result

This option displays the full path of the verification results folder associated with the current test run.

You can use this option with the **getvar** function.

You can also view the full path of the verification results folder from the corresponding **Verification results folder** box in the **Current Test** tab of the Test Properties dialog box as described in Chapter 22, "Setting Properties for a Single Test" in the *Mercury WinRunner Basic Features User's Guide*.

runmode

This option displays the current run mode.

You can use this option with the **getvar** function.

Possible values: verify, debug, update (text strings)

You can also view the current run mode from the corresponding **Run mode** box in the **Current Test** tab of the Test Properties dialog box, described in Chapter 22, "Setting Properties for a Single Test" in the *Mercury WinRunner Basic Features User's Guide*.

searchpath

This option sets the path(s) in which WinRunner searches for called tests. If you define search paths, you do not need to designate the full path of a test in a call statement. You can set multiple search paths in a single statement by leaving a space between each path. To set multiple search paths for long file names, surround each path with angle brackets < >. WinRunner searches for a called test in the order in which multiple paths appear in the **getvar** or **setvar** statement.

You can use this option with the **setvar** and **getvar** functions.

You can also set this option using the corresponding **Search path for called tests** box in the **Folders** category of the General Options dialog box as described in Chapter 23, "Setting Global Testing Options" in the *Mercury WinRunner Basic Features User's Guide*.

Note that you can also set this option using the corresponding -*search_path* command line option, described in Chapter 15, "Running Tests from the Command Line."

Note: When WinRunner is connected to Quality Center, you can specify the paths in a Quality Center database that WinRunner searches for called tests. Search paths in a Quality Center database can be preceded by [QC].

---

shared_checklist_dir

This option designates the folder in which WinRunner stores shared checklists for GUI and database checkpoints. In the test script, shared checklist files are designated by SHARED_CL before the file name in a **win_check_gui**, **obj_check_gui**, **check_gui**, or **check_db** statement. For more information on shared GUI checklists, refer to Chapter 9, "Checking GUI Objects" in the *Mercury WinRunner Basic Features User's Guide*. For more information on shared database checklists, refer to Chapter 14, "Checking Databases" in the *Mercury WinRunner Basic Features User's Guide*. Note that if you designate a new folder, you must restart WinRunner in order for the change to take effect.

You can use this option with the **getvar** function.

You can also view the location of the folder in which WinRunner stores shared checklists from the corresponding **Shared checklists** box in the **Folders** category of the General Options dialog box as described in Chapter 23, "Setting Global Testing Options" in the *Mercury WinRunner Basic Features User's Guide*.

silent_mode

This option displays whether WinRunner is running in silent mode. In silent mode, WinRunner suppresses messages during a test run so that a test can run unattended. When you run a test remotely from Quality Center, you must run it in silent mode, because no one is monitoring the computer where the test is running to view the messages. For information on running tests remotely from Quality Center, see Chapter 26, "Managing the Testing Process."

You can use this option with the **setvar** and **getvar** functions.

Possible values: on, off (text strings)

Note: When you run tests in batch mode, you automatically run them in silent mode. For information running tests in batch mode, see Chapter 14, "Running Batch Tests."

single_prop_check_fail

This option fails a test run when **_check_info** statements fail. It also writes an event to the Test Results window for these statements. (You can create **_check_info** statements using the **Insert** > **GUI Checkpoint** > **For Single Property** command.)

You can use this option with the **setvar** and **getvar** functions.

Possible values: 1,0

For information about the **check_info** functions, refer to the *TSL Reference*.

You can also set this option using the corresponding **Fail test when single property check fails** option in the **Run** > **Settings** category of the General Options dialog box as described in Chapter 23, “Setting Global Testing Options” in the *Mercury WinRunner Basic Features User’s Guide*.

Note that you can also set this option using the corresponding -*single_prop_check_fail* command line option, described in Chapter 15, “Running Tests from the Command Line.”

speed

This option sets the default run *s*peed for tests run in Analog mode.

Possible values: normal, fast (text strings)

Setting the option to **normal** runs the test at the speed at which it was recorded.

Setting the option to **fast** runs the test as fast as the application can receive input.

You can use this option with the **setvar** and **getvar** functions.

You can also set this option using the corresponding **Run speed for Analog mode** option in the **Run** category of the General Options dialog box as described in Chapter 23, “Setting Global Testing Options” in the *Mercury WinRunner Basic Features User’s Guide*.

Note that you can also set this option using the corresponding -*speed* command line option, described in Chapter 15, “Running Tests from the Command Line.”

sync_fail_beep

This option determines whether WinRunner beeps when synchronization fails.

You can use this option with the **setvar** and **getvar** functions.

Possible values: 1,0

You can also set this option using the corresponding **Beep when synchronization fails** check box in the **Run** > **Synchronization** category of the General Options dialog box as described in Chapter 23, "Setting Global Testing Options" in the *Mercury WinRunner Basic Features User's Guide*.

Note: This option is useful primarily for debugging test scripts.

Note: If synchronization often fails during your test runs, consider increasing the value of the *synchronization_timeout* testing option (described below) or the corresponding **Timeout for waiting for synchronization message** option in the **Run** > **Synchronization** category of the General Options dialog box.

synchronization_timeout

This option sets the timeout (in milliseconds) that WinRunner waits before validating that keyboard or mouse input was entered correctly during a test run.

You can use this option with the **setvar** and **getvar** functions.

Possible values: numbers 0 and higher

You can also set this option using the corresponding **Timeout for waiting for synchronization message** box in the **Run** > **Synchronization** category of the General Options dialog box as described in Chapter 23, "Setting Global Testing Options" in the *Mercury WinRunner Basic Features User's Guide*.

Note: If synchronization often fails during your test runs, consider increasing the value of this option.

qc_connection

This option indicates whether WinRunner is currently connected to Quality Center. (Formerly *td_connection* or *test_director.*)

You can use this option with the **getvar** function.

Possible values: on, off (text strings)

You can connect to Quality Center from the Quality Center Connection dialog box or using the *-qc_connection* command line option. For more information about connecting to Quality Center, see Chapter 26, "Managing the Testing Process."

qc_cycle_name

This option displays the name of the Quality Center test set (formerly known as "cycle") for the test. (Formerly *td_cycle_name* or *cycle*.)

You can use this option with the **getvar** function.

You can set this option using the Run Tests dialog box when you run a test set from WinRunner while connected to Quality Center. For more information, see "Running Tests in a Test Set" on page 402. You can also set this option from within Quality Center. For more information, refer to the *Mercury Quality Center User's Guide*.

Note that you can also set this option using the corresponding *-qc_cycle_name* command line option, described in Chapter 15, "Running Tests from the Command Line."

qc_database_name

This option displays the name of the Quality Center project database to which WinRunner is currently connected. (Formerly *td_database_name*)

You can use this option with the **getvar** function.

You can set this option using the **Project** option in the **Quality Center Connection** dialog box, which you can open by choosing **Tools** > **Quality Center Connection**. For more information, see Chapter 26, "Managing the Testing Process."

Note that you can also set this option using the corresponding -*qc_database_name* command line option, described in Chapter 15, "Running Tests from the Command Line."

qc_server_name

This option displays the name of the Quality Center server to which WinRunner is currently connected. (Formerly *td_server_name)*

You can use this option with the **getvar** function.

You can set this option using the **Server** box in the Quality Center Connection dialog box, which you can open by choosing **Tools** > **Quality Center Connection**. For more information, see Chapter 26, "Managing the Testing Process."

Note that you can also set this option using the corresponding -*qc_server_name* command line option, described in Chapter 15, "Running Tests from the Command Line."

qc_test_instance

This option displays the instance of the test that is currently opened and running in the Quality Center test set.

You can use this option with the **getvar** function.

You can set this value using the **Test Instance** box in the Test Run dialog box when you are connected to Quality Center. For more information, see "Running Tests in a Test Set" on page 402.

qc_test_run_id

This option displays the run name of the test that is currently opened and running in the Quality Center test set.

You can use this option with the **getvar** function.

You can set this value using the **Test Run Name** box in the Test Run dialog box when you are connected to Quality Center. For more information, see "Running Tests in a Test Set" on page 402.

qc_user_name

This option displays the user name for opening the selected Quality Center database. (Formerly *td_user_name* or *user*.)

You can use this option with the **getvar** function.

Note that you can also set this option using the corresponding *-qc_user_name* command line option, described in Chapter 15, “Running Tests from the Command Line.”

You can set this option using the **User name** box in the Quality Center Connection dialog box, which you can open by choosing **Tools** > **Quality Center Connection**. For more information, see Chapter 26, “Managing the Testing Process.”

tempdir

This option designates the folder containing temporary files. Note that if you designate a new folder, you must restart WinRunner in order for the change to take effect.

You can use this option with the **setvar** and **getvar** functions.

You can also set this option using the corresponding **Temporary files** box in the **Folders** category of the General Options dialog box as described in Chapter 23, “Setting Global Testing Options” in the *Mercury WinRunner Basic Features User’s Guide*.

testname

This option displays the full path of the current test.

You can use this option with the **getvar** function.

You can also view the location and the test name of the current test in the **General** tab of the Test Properties dialog box as described in Chapter 22, “Setting Properties for a Single Test” in the *Mercury WinRunner Basic Features User’s Guide*.

timeout_msec

This option sets the global timeout (in milliseconds) used by WinRunner when executing checkpoints and Context Sensitive statements. This value is added to the *time* parameter embedded in GUI checkpoint or synchronization point statements to determine the maximum amount of time that WinRunner searches for the specified window. The timeout must be greater than the delay for window synchronization (as set with the *delay_msec* testing option). This option was formerly known as *timeout*, and was measured in seconds.

For example, in the statement:

```
win_check_bitmap ("calculator", Img1, 2, 261,269,93,42);
```

when the *timeout_msec* variable is 10,000 milliseconds, this operation takes a maximum of 12,000 (2,000 +10,000) milliseconds.

You can use this option with the **setvar** and **getvar** functions.

Possible values: numbers 0 and higher

Note: This option is accurate to within 20-30 milliseconds.

You can also set this option using the corresponding **Timeout for checkpoints and CS statements** box in the **Run** > **Settings** category of the General Options dialog box as described in Chapter 23, “Setting Global Testing Options” in the *Mercury WinRunner Basic Features User’s Guide*.

Note that you can also set this option using the corresponding *-timeout_msec* command line option, described in Chapter 15, “Running Tests from the Command Line.”

Treeview_path_separator

This option defines the string recorded in the test script to separate items in a tree view path.

Possible values: any text string

You can use this option with the **getvar** and **setvar** functions.

You can also set this option using the corresponding **String for parsing a TreeView path** box in the **Record** > **Script Format** category of the General Options dialog box as described in Chapter 23, “Setting Global Testing Options” in the *Mercury WinRunner Basic Features User’s Guide*.

22

Customizing the Function Generator

You can customize the Function Generator to include the user-defined functions that you most frequently use in your tests scripts. This makes programming tests easier and reduces the potential for errors.

This chapter describes:

- About Customizing the Function Generator
- Adding a Category to the Function Generator
- Adding a Function to the Function Generator
- Associating a Function with a Category
- Adding a Subcategory to a Category
- Setting a Default Function for a Category

About Customizing the Function Generator

You can modify the Function Generator to include the user-defined functions that you use most frequently. This enables you to quickly generate your favorite functions and insert them directly into your test scripts. You can also create custom categories in the Function Generator in which you can organize your user-defined functions. For example, you can create a category named **my_button**, which contains all the functions specific to the **my_button** custom class. You can also set the default function for the new category, or modify the default function for any standard category.

To add a new category with its associated functions to the Function Generator:

1. Add a new category to the Function Generator.
2. Add new functions to the Function Generator.
3. Associate the new functions with the new category.
4. Set the default function for the new category.
5. Add a subcategory for the new category (optional).

You can find all the functions required to customize the Function Generator in the “function table” category of the Function Generator. By inserting these functions in a startup test, you ensure that WinRunner is invoked with the correct configuration.

Adding a Category to the Function Generator

You use the **generator_add_category** TSL function to add a new category to the Function Generator. This function has the following syntax:

generator_add_category (*category_name* **);**

where *category_name* is the name of the category that you want to add to the Function Generator.

In the following example, the **generator_add_category** function adds a category called “my_button” to the Function Generator:

```
generator_add_category ("my_button");
```

Note: If you want to display the default function for category when you select an object using the **Insert** > **Function** > **For Object/Window** command, the category name must be the same as the name of the GUI object class.

To add a category to the Function Generator:

1. Open the Function Generator. (Choose **Insert** > **Function** > **From Function Generator**, click the **Insert Function from Function Generator** button on the User toolbar, or press the INSERT FUNCTION FROM FUNCTION GENERATOR softkey.)
2. In the **Category** box, click **function table**.
3. In the **Function Name** box, click **generator_add_category**.
4. Click **Args**. The Function Generator expands.
5. In the **Category Name** box, type the name of the new category between the quotes. Click **Paste** to paste the TSL statement into your test script.
6. Click **Close** to close the Function Generator.

A **generator_add_category** statement is inserted into your test script.

Note: You must run the test script in order to insert a new category into the Function Generator.

Adding a Function to the Function Generator

When you add a function to the Function Generator, you specify the following:

- how the user supplies values for the arguments in the function
- the function description that appears in the Function Generator

Note that after you add a function to the Function Generator, you should associate the function with a category. See “Associating a Function with a Category” on page 323.

You use the **generator_add_function** TSL function to add a user-defined function to the Function Generator.

To add a function to the Function Generator:

1. Open the Function Generator. (Choose **Insert** > **Function** > **From Function Generator**, click the **Insert Function from Function Generator** button on the User toolbar, or press the INSERT FUNCTION FROM FUNCTION GENERATOR softkey.)

2. In the **Category** box, click **function table**.

3. In the **Function Name** box, click **generator_add_function**.

4. Click **Args**. The Function Generator expands.

5. In the Function Generator, define the *function_name*, *description*, and *arg_number* arguments:

   - In the *function_name* box, type the name of the new function between the quotes. Note that you can include spaces and upper-case letters in the function name.

   - In the *description* box, enter the description of the function between the quotes. Note that it does not have to be a valid string expression and it must not exceed 180 characters.

   - In the *arg_number* box, you must choose 1. To define additional arguments (up to eight arguments for each new function), you must manually modify the generated **generator_add_function** statement once it is added to your test script.

6. For the function's first argument, define the following arguments: *arg_name*, *arg_type*, and *default_value* (if relevant):

   - In the *arg_name* box, type the name of the argument within the quotation marks. Note that you can include spaces and upper-case letters in the argument name.

   - In the *arg_type* box, select "***browse()***", "***point_object", "point_window***", "***select_list (01)***", or "***type_edit***", to choose how the user will fill in the argument's value in the Function Generator, as described in "Defining Function Arguments" on page 317.

   - In the *default_value* box, if relevant, choose the default value for the argument.

- Note that any additional arguments for the new function cannot be added from the Function Generator: The *arg_name*, *arg_type*, and *default_value* arguments must be added manually to the **generator_add_function** statement in your test script.

7 Click **Paste** to paste the TSL statement into your test script.

8 Click **Close** to close the Function Generator.

Note: You must run the test script in order to insert a new function into the Function Generator.

Defining Function Arguments

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

generator_add_function (*function_name, description, arg_number,*
  arg_name_1, arg_type_1, default_value_1,
      ...
  arg_name_n, arg_type_n, default_value_n **);**

- *function_name* is the name of the function you are adding.
- *description* is a brief explanation of the function. The description appears in the Description box of the Function Generator when the function is selected. It does not have to be a valid string expression and must not exceed 180 characters.
- *arg_number* is the number of arguments in the function. This can be any number from zero to eight.

For each argument in the function you define, you supply the name of the argument, how it is filled in, and its default value (where possible). When you define a new function, you repeat the following parameters for each argument in the function: *arg_name, arg_type,* and *default_value.*

- *arg_name* defines the name of the argument that appears in the Function Generator.

➤ *arg_type* defines how the user fills in the argument's value in the Function Generator. There are five types of arguments:

"browse()": The value of the argument is evaluated by pointing to a file in a browse file dialog box. Use *browse* when the argument is a file. To select a file with specific file extensions only, specify a list of default extension(s). Items in the list should be separated by a space or tab. Once a new function is defined, the *browse* argument is defined in the Function Generator by using a Browse button.

"point_object": The value of the argument is evaluated by pointing to a GUI object (other than a window). Use *point_object* when the argument is the logical name of an object. Once a new function is defined, the *point_object* argument is defined in the Function Generator by using a pointing hand.

"point_window": The value of the argument is evaluated by pointing to a window. Use *point_window* when the argument is the logical name of a window. Once a new function is defined, the *point_window* argument is defined in the Function Generator by using a pointing hand.

"select_list (01)": The value of the argument is selected from a list. Use *select_list* when there is a limited number of argument values, and you can supply all the values. Once a new function is defined, the *select_list* argument is defined in the Function Generator by using a combo box.

"type_edit": The value of the argument is typed in. Use *type_edit* when you cannot supply the full range of argument values. Once a new function is defined, the *type_edit* argument is defined in the Function Generator by typing into an edit field.

- *default_value* provides the argument's default value. You may assign default values to **select_list** and **type_edit** arguments. The default value you specify for a **select_list** argument must be one of the values included in the list. You cannot assign default values to **point_window** and **point_object** arguments.

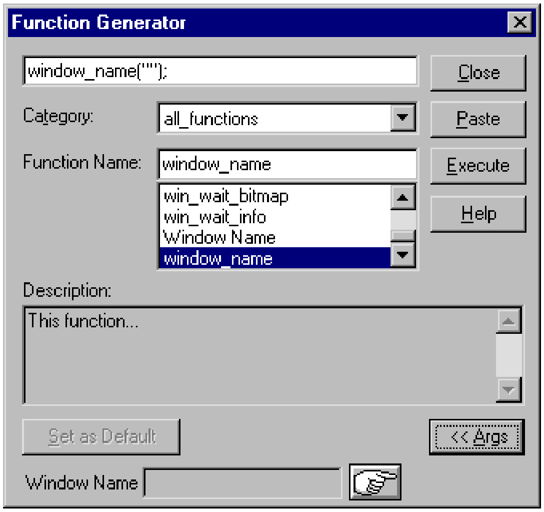
The following are examples of argument definitions that you can include in **generator_add_function** statements. The examples include the syntax of the argument definitions, their representations in the Function Generator, and a brief description of each definition.

Example 1

```
generator_add_function ("window_name","This function...",1,
    "Window Name","point_window","");
```

The *function_name* is window_name. The *description* is "This function...". The *arg_number* is 1. The *arg_name* is Window Name. The *arg_type* is point_window. There is no *default_value*: since the argument is selected by pointing to a window, this argument is an empty string.

When you select the **window_name** function in the Function Generator and click the **Args** button, the Function Generator appears as follows:



Function Generator

window_name("");

Category: all_functions

Function Name: window_name

win_wait_bitmap
win_wait_info
Window Name
window_name

Description:

This function...

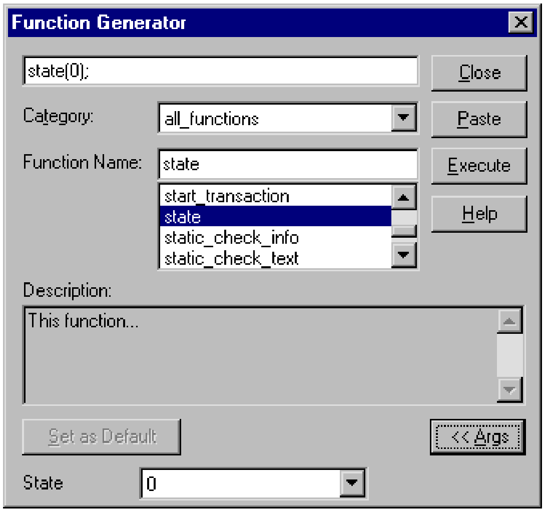
Window Name

Example 2

generator_add_function("state","This function...",1,"State","select_list (0 1)",0);

The *function_name* is state. The *description* is “This function...”. The *arg_number* is 1. The *arg_name* is State. The *arg_type* is select_list. The *default_value* is 0.

When you select the **state** function in the Function Generator and click the **Args** button, the Function Generator appears as follows:



Function Generator

state(0);

Category: all_functions

Function Name: state

start_transaction
state
static_check_info
static_check_text

Description:

This function...

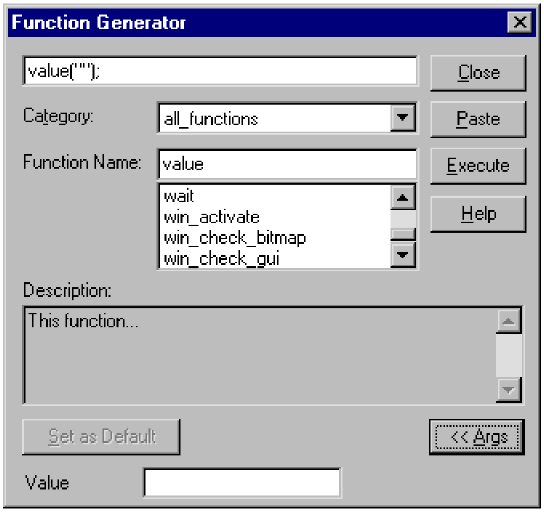
State 0

Example 3

generator_add_function("value","This function...",1,"Value","type_edit","");

The *function_name* is value. The *description* is “This function...”. The *arg_number* is 1. The *arg_name* is Value. The *arg_type* is type_edit. There is no *default_value*.

When you select the **value** function in the Function Generator and click the **Args** button, the Function Generator appears as follows:



Defining Property Arguments

You can define a function with an argument that uses a Context Sensitive property, such as the label on a pushbutton or the width of a checkbox. In such a case, you cannot define a single default value for the argument. However, you can use the **attr_val** function to determine the value of a property for the selected window or GUI object. You include the **attr_val** function in a call to the **generator_add_function** function.

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

attr_val (*object_name, "property"* **);**

- *object_name* defines the window or GUI object whose property is returned. It must be identical to the *arg_name* defined in a previous argument of the **generator_add_function** function.
- *property* can be any property used in Context Sensitive testing, such as height, width, label, or value. You can also specify platform-specific properties such as MSW_class and MSW_id.

You can either define a specific property, or specify a parameter that was defined in a previous argument of the same call to the function, **generator_add_function**. For an illustration, see example 2, below.

Example 1

In this example, a function called “check_my_button_label” is added to the Function Generator. This function checks the label of a button.

```
generator_add_function("check_my_button_label", "This function checks the
label of a button.", 2,
    "button_name", "point_object"," ",
    "label", "type_edit", "attr_val(button_name, \"label\")");
```

The “check_my_button_label” function has two arguments. The first is the name of the button. Its selection method is *point_object* and it therefore has no default value. The second argument is the label property of the button specified, and is a *type_edit* argument. The **attr_val** function returns the label property of the selected GUI object as the default value for the property.

Example 2

The following example adds a function called “check_my_property” to the Function Generator. This function checks the *class*, *label*, or *active* property of an object. The property whose value is returned as the default depends on which property is selected from the list.

```
generator_add_function ("check_my_property","This function checks an object’s
property.",3,
    "object_name", "point_object", " ",
    "property", "select_list(\"class\"\"label\"\"active\")", "\"class\"",
    "value:", "type_edit", "attr_val(object_name, property)");
```

The first three arguments in **generator_add_function** define the following:

➤ the name of the new function (check_my_property).

➤ the description appearing in the Description field of the Function Generator. This function checks an object’s property.

➤ the number of arguments (3).

The first argument of “check_my_property” determines the object whose property is to be checked. The first parameter of this argument is the object name. Its type is *point_object*. Consequently, as the null value for the third parameter of the argument indicates, it has no default value.

The second argument is the property to be checked. Its type is *select_list*. The items in the list appear in parentheses, separated by field separators and in quotation marks. The default value is the class property.

The third argument, value, is a *type_edit* argument. It calls the **attr_val** function. This function returns, for the object defined as the function’s first argument, the property that is defined as the second argument (class, label or active).

Associating a Function with a Category

You should associate any function that you add to the Function Generator with an existing category. You make this association using the **generator_add_function_to_category** TSL function. Both the function and the category must already exist.

This function has the following syntax:

generator_add_function_to_category (*category_name*, *function_name* **);**

- *category_name* is the name of a category in the Function Generator. It can be either a standard category, or a custom category that you defined using the **generator_add_category** function.
- *function_name* is the name of a custom function. You must have already added the function to the Function Generator using the function, **generator_add_function**.

To associate a function with a category:

1. Open the Function Generator. (Choose **Insert** > **Function** > **From Function Generator**, click the **Insert Function from Function Generator** button on the User toolbar, or press the INSERT FUNCTION FROM FUNCTION GENERATOR softkey.)
2. In the **Category** box, click **function table**.

3 In the **Function Name** box, click **generator_add_function_to_category**.

4 Click **Args**. The Function Generator expands.

5 In the **Category Name** box, enter the category name as it already appears in the Function Generator.

6 In the **Function Name** box, enter the function name as it already appears in the Function Generator.

7 Click **Paste** to paste the TSL statement into your test script.

8 Click **Close** to close the Function Generator.

A **generator_add_function_to_category** statement is inserted into your test script. In the following example, the "check_my_button_label" function is associated with the "my_button" category. This example assumes that you have already added the "my_button" category and the "check_my_button_label" function to the Function Generator.

```
generator_add_function_to_category ("my_button", "check_my_button_label");
```

Note: You must run the test script in order to associate a function with a category.

Adding a Subcategory to a Category

You use the **generator_add_subcategory** TSL function to make one category a subcategory of another category. Both categories must already exist. The **generator_add_subcategory** function adds all the functions in the subcategory to the list of functions for the parent category.

If you create a separate category for your new functions, you can use the **generator_add_subcategory** function to add the new category as a subcategory of the relevant Context Sensitive category.

The syntax of **generator_add_subcategory** is as follows:

generator_add_subcategory (*category_name*, *subcategory_name* **);**

➤ *category_name* is the name of an existing category in the Function Generator.

➤ *subcategory_name* is the name of an existing category in the Function Generator.

To add a subcategory to a category:

1. Open the Function Generator. (Choose **Insert** > **Function** > **From Function Generator**, click the **Insert Function from Function Generator** button on the User toolbar, or press the INSERT FUNCTION FROM FUNCTION GENERATOR softkey.)
2. In the **Category** box, click **function table**.
3. In the **Function Name** box, click **generator_add_subcategory**.
4. Click **Args**. The Function Generator expands.
5. In the **Category Name** box, enter the category name as it already appears in the Function Generator.
6. In the **Subcategory Name** box, enter the subcategory name as it already appears in the Function Generator.
7. Click **Paste** to paste the TSL statement into your test script.
8. Click **Close** to close the Function Generator.

A **generator_add_subcategory** statement is inserted into your test script. In the following example, the “my_button” category is defined as a subcategory of the “push_button” category. All “my_button” functions are added to the list of functions defined for the push_button category.

```
generator_add_subcategory ("push_button", "my_button");
```

Note: You must run the test script in order to add a subcategory to a category.

Setting a Default Function for a Category

You set the default function for a category using the **generator_set_default_function** TSL function. This function has the following syntax:

generator_set_default_function (*category_name*, *function_name* **);**

- *category_name* is an existing category.
- *function_name* is an existing function.

You can set a default function for a standard category or for a user-defined category that you defined using the **generator_add_category** function. If you do not define a default function for a user-defined category, WinRunner uses the first function in the list as the default function.

Note that the **generator_set_default_function** function performs the same operation as the Set As Default button in the Function Generator dialog box. However, a default function set through the Set As Default checkbox remains in effect during the current WinRunner session only. By adding **generator_set_default_function** statements to your startup test, you can set default functions permanently.

To set a default function for a category:

1 Open the Function Generator. (Choose **Insert** > **Function** > **From Function Generator**, click the **Insert Function from Function Generator** button on the User toolbar, or press the INSERT FUNCTION FROM FUNCTION GENERATOR softkey.)

2 In the **Category** box, click **function table**.

3 In the **Function Name** box, click **generator_set_default_function.**

4 Click **Args**. The Function Generator expands.

5 In the **Category Name** box, enter the category name as it already appears in the Function Generator.

6 In the **Default** box, enter the function name as it already appears in the Function Generator.

7 Click **Paste** to paste the TSL statement into your test script.

8 Click **Close** to close the Function Generator.

A **generator_set_default_function** statement is inserted into your test script. In the following example, the default function of the push button category is changed from **button_check_enabled** to the user-defined “check_my_button_label” function.

```
generator_set_default_function ("push_button", "check_my_button_label");
```

Note: You must run the test script in order to set a default function for a category.

23

Initializing Special Configurations

By creating *startup tests*, you can automatically initialize special testing configurations each time you start WinRunner.

This chapter describes:

- About Initializing Special Configurations
- Creating Startup Tests
- Sample Startup Test

About Initializing Special Configurations

A startup test is a test script that is automatically run each time you start WinRunner. You can create startup tests that load GUI map files and compiled modules, configure recording, and start the application under test.

You designate a test as a startup test by entering its location in the **Startup test** box in the **General** > **Startup** category in the General Options dialog box. For more information on using the General Options dialog box, refer to Chapter 23, "Setting Global Testing Options" in the *Mercury WinRunner Basic Features User's Guide*.

Creating Startup Tests

You should add the following types of statements to your startup test:

- **load** statements, which load compiled modules containing user-defined functions that you frequently call from your test scripts.
- **GUI_load** statements, which load one or more GUI map files. This ensures that WinRunner recognizes the GUI objects in your application when you run tests.
- statements that configure how WinRunner records GUI objects in your application, such as **set_record_attr** or **set_class_map.**
- an **invoke_application** statement, which starts the application being tested.
- statements that enable WinRunner to generate custom record TSL functions when you perform operations on custom objects, such as **add_cust_record_class**.

By including the above elements in a startup test, WinRunner automatically compiles all designated functions, loads all necessary GUI map files, configures the recording of GUI objects, and loads the application being tested.

Note: You can use the RapidTest Script wizard to create a basic startup test called *myinit* that loads a GUI map file and the application being tested. Note that when you work in the *GUI Map File per Test* mode (described in Chapter 6, “Working in the GUI Map File per Test Mode” in the *Mercury WinRunner Basic Features User’s Guide*) the *myinit* test does not load GUI map files.

Sample Startup Test

The following is an example of the types of statements that might appear in a startup test:

```
# Start the Flight application if it is not already displayed on the screen
if ((rc=win_exists("Flight")) == E_NOT_FOUND)
    invoke_application("w:\\flight_app\\flight.exe", "", "w:\\flight_app",
SW_SHOW);

# Load the compiled module "qa_funcs"
load("qa_funcs", 1, 1);

# Load the GUI map file "flight.gui"
GUI_load ("w:\\qa\\gui\\flight.gui");

# Map the custom “borbtn” class to the standard “push_button” class
set_class_map (“borbtn”, “push_button”);
```

Part VII

Working with Other Mercury Products

24

Working with Business Process Testing

Business Process Testing is a module of Mercury Quality Center that utilizes a new methodology for quality-assurance testing, based on the creation and implementation of components in business process tests. This methodology enables non-technical SMEs (subject matter experts) to design business process tests early in the development cycle and in a script-free environment.

Integrating WinRunner with Business Process Testing enables you to leverage your investment in existing WinRunner scripts and improve the test automation process by using the Business Process Testing framework.

This chapter describes how to use WinRunner to create and manage scripted components that are used in Business Process Testing. WinRunner options and features that are common or similar for both components and tests are described in the relevant chapters throughout this user's guide.

This chapter describes:

About Business Process Testing

A business process test is a scenario composed of a serial flow of components, which are easily-maintained reusable scripts that perform a specific task in the application being tested.

Generally, components are created and modified in Quality Center by SMEs. However, when WinRunner is connected to a Quality Center project with Business Process Testing, you can create, view, modify, and debug components in WinRunner. You can then save your components to a project in Quality Center in the form of scripted components. SMEs working in Quality Center can combine each of your components into one or more business process tests.

The SME can combine components from other testing tools, such as QuickTest, with WinRunner components in business process tests, ensuring that every aspect of the application to be tested is covered, even before it is ready to be tested.

You can also save your existing WinRunner test scripts as scripted components, so that SMEs can use them in business process tests.

Note: You can integrate Business Process Testing with WinRunner by purchasing Business Process Testing licenses. In addition, to work with Business Process Testing from within WinRunner, you must connect to a Quality Center project with Business Process Testing support.

You can add specially formulated comments to your WinRunner components to describe their steps. (Steps represent operations to be performed during the business process test run.) The SME can view these comments in Quality Center and can use them to understand the flow of the script.

SMEs can view and work with scripted components in Quality Center modules. However, they cannot modify scripted component steps.

The remaining sections in this chapter describe options and features that are unique to working with scripted components in WinRunner. For more information on scripted components, refer to the *Business Process Testing User's Guide*.

Understanding Business Process Testing Methodology

Components are parts of a business process that has been broken down into smaller parts. Components are the building blocks from which an effective business process testing structure can be produced.

For example, in most applications users need to log in before they can do anything else. A WinRunner expert can create one scripted component that tests the login procedure for an application. This component can then be reused in multiple business process tests, resulting in easier maintenance, updating, and test management.

Components are comprised of steps. For example, the login component's first step may be to open the application. Its second step could be entering a user name. Its third step could be entering a password, and its fourth step could be clicking the **Enter** button.

You create scripted components in WinRunner by recording steps or by manually entering steps on applications designed in any supported environment. You can add checkpoints and output values, parameterize selected items, and enhance the component with flow statements and other testing functions. You then save the scripted component to a project in Quality Center. An SME using Business Process Testing in Quality Center combines your saved components into one or more business process tests, which are used to check that the application behaves as expected.

The component creation process can be divided into two elements: the component shell and the component implementation.

- The *component shell* is the component's outer layer. The information in the shell is visible or available at the test level. The component shell information can be created in WinRunner or Quality Center.

  Once the component shell is created, it can be used by the SME to build business process tests even if the script implementation has not yet begun.

- The *component implementation* is the component's inner layer. It includes the actual script and specific settings for the component. The information can be seen only at the component level. You create the component implementation using WinRunner.

Understanding the Differences Between Components and Tests

If you are already familiar with using WinRunner to create tests, you will find that the procedures for creating and editing scripted components are quite similar. However, due to the design and purpose of the component model, there are certain differences in the way you design, edit, and run components. The guidelines in the sections below provide an overview of these differences.

General Differences Between Components and Tests

Following are guidelines and information regarding differences between components and tests:

- A component can call another WinRunner component, but it cannot call a WinRunner test, a QuickTest test, a business process test, or a QuickTest component.
- A WinRunner test can call another test, but it cannot call a WinRunner component, a keyword-driven component, or a business process test.
- When working with components, all external files are stored in the Quality Center project to which you are currently connected.
- When working with components, only the **Debug** and **Update** run modes are supported. The **Verify** run mode is not available to run components because verification that the application works is performed when the component is run as part of a business process test in Quality Center.

- Specific toolbar and menu commands are used to create and open tests and components. You can use the **New** and **Open** toolbar buttons to create or open tests. You can use the menu commands to create and open tests or components.
- By default, the **Save** command and toolbar button save untitled documents as tests. However, if you work with components, you can change this default so that the **Save** command and toolbar button save untitled documents as components. For more information, see “Setting WinRunner Scripted Component as the Default Save Type” on page 368.

Differences When Using the Data Table with Components

You should consider the following when using the Data Table with components.

- If you enter and parameterize test data values in more than one row of the Data Table, then each component iteration run will perform the relevant data table loop according to the number of rows in the data table (in addition to component iterations according to the data set for the component parameters).
- The component (or test saved in Quality Center) can use any Quality Center Data Table according to the specified Quality Center path. If a Data Table is not saved with the test, the user must upload the Data Table to the Quality Center project.

For more information on Data Tables, refer to Chapter 18, “Creating Data-Driven Tests” in the *Mercury WinRunner Basic Features User’s Guide*.

Understanding Business Process Testing Roles

The Business Process Testing model is role-based, allowing non-technical SMEs to define and document components and business process tests. Automation Engineers for testing tool applications such as WinRunner, record, program, and debug individual steps of components, which SMEs can include in their business process tests.

Note: The role structure and the tasks performed by various roles in your organization may differ from those described here, according to the methodology adopted by your organization. For example, the tasks of the SME and the testing tool engineer may be performed by the same person.

The following basic user roles are identified in the Business Process Testing model when working with WinRunner:

➤ **Subject Matter Expert (SME)—**The SME has specific knowledge of the application logic, a high-level understanding of the entire system, and a detailed understanding of the individual elements and tasks that are fundamental to the application being tested.

This enables the SME to determine the operating scenarios or business processes that must be tested and to identify the key business activities that are common to multiple business processes. The SME is also responsible for maintaining the testing steps for each of the individual components created within Quality Center.

One of the great advantages of the Business Process Testing model is that the work of the SME is not dependent on the completion of component implementation by the Automation Engineer. Hence, using Business Process Testing, the testing process can start early in the development cycle, before the application to be tested is at a level at which automated components can be recorded.

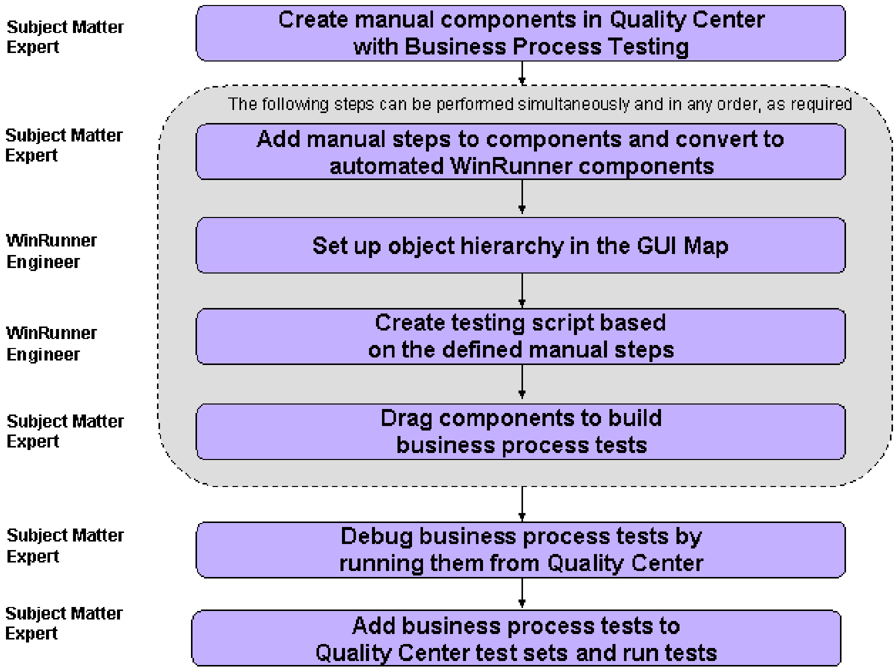
The SME configures the values used for business process tests, runs them in test sets, and reviews the results.

- **Automation Engineer—**The Automation Engineer is an expert for an automated testing tool, such as WinRunner. The Automation Engineer is responsible for implementing, maintaining, and debugging the scripted components created within WinRunner.

Note: This chapter is primarily targeted at the Automation Engineer and the tasks that the Automation Engineer can perform in WinRunner. For more information on the options available to the SME in Quality Center, refer to the *Mercury Quality Center User's Guide*.

Understanding the Business Process Testing Workflow

The following is an example of a common Business Process Testing workflow. The actual workflow in an organization may differ for different projects, or at different stages of the product development life cycle.


Subject Matter Expert
Create manual components in Quality Center with Business Process Testing
The following steps can be performed simultaneously and in any order, as required
Subject Matter Expert
Add manual steps to components and convert to automated WinRunner components
WinRunner Engineer
Set up object hierarchy in the GUI Map
WinRunner Engineer
Create testing script based on the defined manual steps
Subject Matter Expert
Drag components to build business process tests
Subject Matter Expert
Debug business process tests by running them from Quality Center
Subject Matter Expert
Add business process tests to Quality Center test sets and run tests

Understanding Business Process Testing Terminology

The following terminology, specific to Business Process Testing, is used in this chapter:

Steps—A step represents an operation to be performed during the business process test run.

Scripted Component (or **Component**)**—**An easily-maintained, reusable unit comprising one or more steps that perform a specific task. Scripted components may require input values from an external source or from other components. They can return output values to other components.

Business Process Test—A scenario comprising a serial flow of components, designed to test a specific business process of an application.

Component Input Parameters—Variable values that a component can receive and use as the values for specific, parameterized steps in the component. A component parameter may be accessed by any component in the Quality Center project.

Component Output Parameters—Values that a component can return. These values can be viewed in the business process test results and can also be used as input for a component that is used later in the test. A component parameter can be accessed by any component in the Quality Center project.

Roles—The various types of users who are involved in Business Process Testing.

Subject Matter Expert (SME)—A person who has specific knowledge of the application logic, a high-level understanding of the entire system, and a detailed understanding of the individual elements and tasks that are fundamental to the application being tested. The Subject Matter Expert uses Quality Center to create components and business process tests.

Automation Engineer—An expert in an automated testing tool, such as WinRunner.

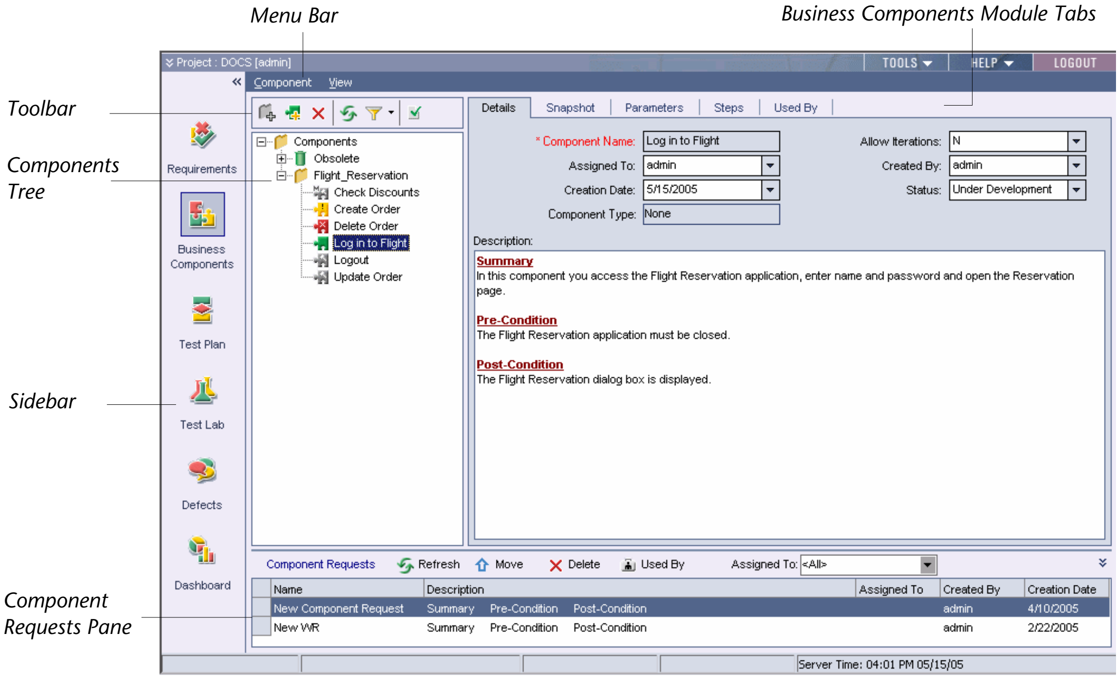
Creating Components in the Quality Center Business Components Module

The SME can create a new component and define its shell and non-automated steps in the Quality Center Business Components module.

The SME can convert the component to a WinRunner component by automating the steps. When non-automated components are converted to WinRunner components, the non-automated steps in the components are displayed in WinRunner as comments, preceded by #'#.

Note: Because WinRunner comments cannot exceed 500 characters, it is important that the SME avoids exceeding 500 characters when creating non-automated steps in Quality Center.

An example of a component in a Quality Center project is shown below:



* Component Name:	Log in to Flight
Assigned To:	admin
Creation Date:	5/15/2005
Component Type:	None
Allow Iterations:	N
Created By:	admin
Status:	Under Development

Description:

Summary
In this component you access the Flight Reservation application, enter name and password and open the Reservation page.

Pre-Condition
The Flight Reservation application must be closed.

Post-Condition
The Flight Reservation dialog box is displayed.

Assigned To: <All>

Name	Description	Assigned To	Created By	Creation Date
New Component Request	Summary Pre-Condition Post-Condition		admin	4/10/2005
New WR	Summary Pre-Condition Post-Condition		admin	2/22/2005

Server Time: 04:01 PM 05/15/05

The component shell includes the following elements:

- **Details**—A general summary of the component's purpose or contents, whether iterations are allowed, and more detailed instructions that define the pre-conditions and the post-conditions for the component.
- **Snapshot**—An image that provides a visual cue or description of the component's purpose or operations.
- **Input parameters—**The name, default value, and description of the data the component can receive.
- **Output parameters—**The name and description of the values that the component can return to the business process test.
- **Status—**The current status of the component, for example, whether the component is fully implemented and ready to be run, or whether it has errors that need to be fixed. The highest severity component status defines the overall status of the business process test.

Note: All the above elements can be created or modified in WinRunner using the Scripted Component Properties dialog box. For more information, see "Defining Scripted Component Properties" on page 352.

Implementing Components in WinRunner

You implement scripted components in WinRunner. You can open and implement an existing component whose shell was defined by the SME, or you can save a WinRunner test as a scripted component in a Quality Center project.

From WinRunner, you implement components by recording steps on any supported environment or by manually programming steps in TSL (WinRunner's Test Script language). You add checkpoints and output values, parameterize selected items, and enhance your test with flow statements and user-defined functions.

From WinRunner you can also view and set options specific to components. For example, you can view the component description, modify the component screenshot, and determine whether component iterations are applicable for the component.

Once a scripted component has been implemented, the SME can open the component in the Quality Center Business Components module to view specially labeled comments that the Automation Engineer enters from WinRunner. These comments should provide a summary of the component steps in the **Steps** tab. The SME or Automation Engineer can also view or debug the component by using Quality Center to launch WinRunner and run the component.

Creating Business Process Tests in the Quality Center Test Plan Module

To create a business process test, the SME selects (drags and drops) the components that apply to the business process test and configures their run settings.

Each component can be used differently by different business process tests. For example, in each test, the component can be configured to use different input parameter values or to run a different number of iterations.

Running Business Process Tests and Analyzing the Results

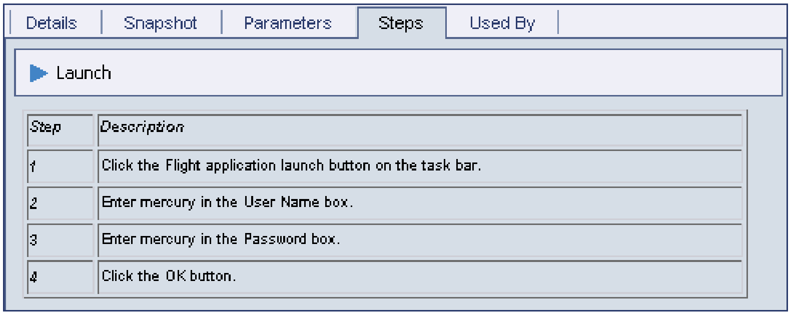
You can use the run and debug options in WinRunner to run and debug an individual scripted component. You can also debug a scripted component within a business process test by running the component from the Test Plan module in Quality Center. Quality Center launches WinRunner and runs the component.

When the business process test has been debugged and is ready for regular test runs, it can be run from the Test Lab module in the same way as any other test is run in Quality Center. Before running the test, the tester can define run-time parameter values and iterations.

From the Test Lab module, the SME can view the results of the entire business process test run. The results include the value of each parameter, and the results of individual events reported by WinRunner.

The Scripted Component View at a Glance

The scripted component is viewed in the Steps tab in the Quality Center Business Components module as a list of step comments. Each comment is displayed in a separate row.



Step	*Description*
1	Click the Flight application launch button on the task bar.
2	Enter mercury in the User Name box.
3	Enter mercury in the Password box.
4	Click the OK button.

For more information, refer to the *Business Process Testing User's Guide*.

Getting Started with Scripted Components in WinRunner

When your Quality Center includes Business Process Testing licences, you can connect WinRunner to your Quality Center project. This enables you to create, view, and debug scripted components that can be included in Quality Center business process tests.

If the Automation Engineer provides easy-to-understand comments in the WinRunner component script, then the SME can view the details of the component and choose to include them in business process tests without the need for any programming knowledge.

Connecting to your Quality Center Project

To work with scripted components, you must first connect WinRunner to a Quality Center server. This server handles the connections between WinRunner and the Quality Center project. Then you choose the Quality Center project that you want WinRunner to access. The project stores component and run session information for the application you are testing. Note that Quality Center projects are password protected, so you must provide a user name and a password.

Note: Before you can work with scripted components in WinRunner, you must enable integration between WinRunner and your Quality Center project. From the main WinRunner window, choose **Tools** > **General Options**. Select the **Run** category. Then select the **Allow other Mercury products to run tests remotely** check box.

For information on how to connect to Quality Center, see “Connecting WinRunner to Quality Center” on page 385.

For information on how to disconnect from Quality Center, see “Disconnecting from Quality Center” on page 387.

Working with Scripted Components

You can utilize the full power of WinRunner tools and options when working with scripted components. For example, you can use the Function Generator to guide you through the process of adding functions to your scripted component. You can also call user-defined functions from compiled modules, parameterize selected items, and add checkpoints and output values to your scripted component.

This chapter describes only how to create scripted components. Use the relevant chapters in this guide for information on how to enter and enhance the steps in your script.

After you create a scripted component, SMEs can view the specially labeled (read-only) comments in the component in the Business Components module of the Quality Center project. They can run the scripted component and add it to their business process tests, but you remain responsible for maintaining the scripted component in WinRunner, if any changes are needed. Scripted components cannot be modified in Quality Center.

You save scripted components in the same way as you save tests. For more information, see “Saving a Scripted Component” on page 363.

Creating a New Scripted Component

This section describes how to create a new scripted component in WinRunner.

Before you create or open a scripted component, you must connect WinRunner to a Quality Center project, which is where scripted components are stored.

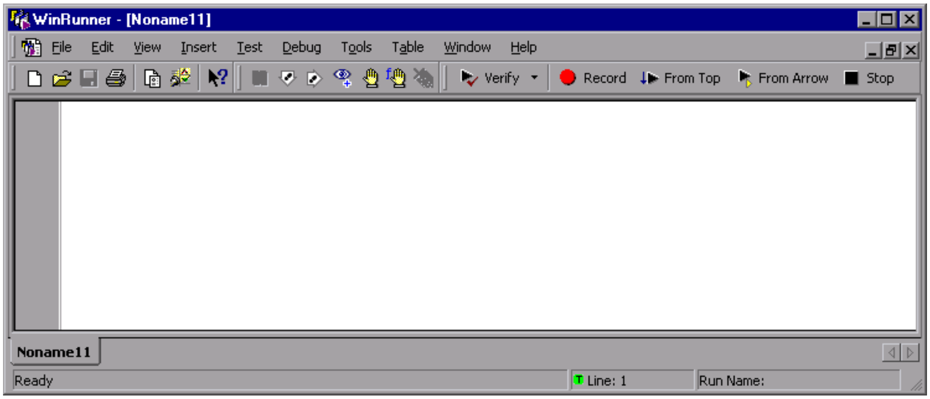
Note: If you want to delete a scripted component, whether it was created in WinRunner or in Quality Center, it can be deleted only from Quality Center. For more information, refer to the *Business Process Testing User's Guide*.

To create a new scripted component:

1 Connect to the Quality Center project in which you want to save the scripted component. For more information, see “Connecting to your Quality Center Project” on page 348.

2 Choose **File** > **New** or press CTRL+N.

A new, untitled test opens.



3 Add steps to the test using the functionality and options provided by WinRunner in the same way as you do for a test. For example, you can use the Step Generator to add steps containing programming logic. You can also add checkpoints and output values to your scripted component.

Note: There are a few testing options that are not available or behave differently when working with scripted components, compared to tests. For more information, see "Understanding the Differences Between Components and Tests" on page 338.

Note: Subject Matter Experts can view read-only comments of scripted components from the Quality Center Business Components module, and they can include them in business process tests using the Test Plan module. They cannot modify the scripted component steps.

4 Define the component's properties, such as description, input and output parameters, status, and whether iterations are allowed, in the Scripted Component Properties dialog box. For more information, see "Defining Scripted Component Properties" on page 352.

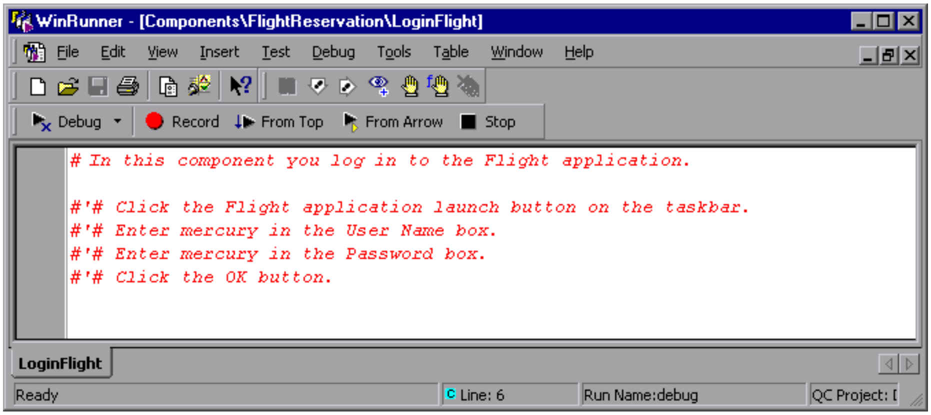
5 Save the test as a scripted component. For more information, see “Saving a Scripted Component” on page 363.

Adding Comments for the SME

WinRunner can provide an HTML presentation of the marked comments that an SME can view in the Business Components module of Quality Center.

For every comment that you want to make visible to the SME in Quality Center, add the prefix #'# to the row.

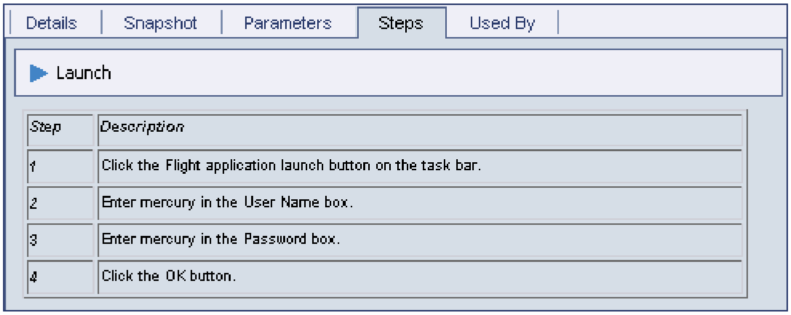
The comments you add in WinRunner could appear as follows:


WinRunner - [Components\FlightReservation\LoginFlight]
File Edit View Insert Test Debug Tools Table Window Help
Debug
Record
From Top
From Arrow
Stop
LoginFlight
Ready
Line: 6
Run Name:debug
QC Project: [

```
# In this component you log in to the Flight application.

#'# Click the Flight application launch button on the taskbar.
#'# Enter mercury in the User Name box.
#'# Enter mercury in the Password box.
#'# Click the OK button.
```

The comments would display in Quality Center as follows:



Step	Description
1	Click the Flight application launch button on the task bar.
2	Enter mercury in the User Name box.
3	Enter mercury in the Password box.
4	Click the OK button.

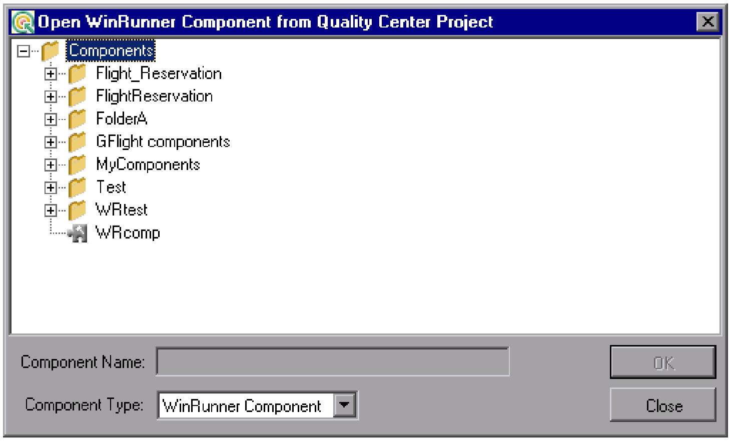
Defining Scripted Component Properties

You can define properties for the scripted component in WinRunner. The SME can view these properties in Quality Center and modify them if required. Conversely, you can view or modify scripted component properties that were originally defined by the SME in Quality Center.

To define scripted component properties:

1 Connect to the Quality Center project that contains the scripted component whose properties you want to define. For more information, see "Connecting to your Quality Center Project" on page 348.

2 Choose **File** > **Open Scripted Component** or press CTRL+H. The Open WinRunner Component from Quality Center Project dialog box opens.



Component Name:

Component Type: WinRunner Component

The status of each component is indicated by its icon. For information on component statuses and their icons, refer to the *Business Process Testing User's Guide*.

Tip: You can open a recently used component by selecting it from the **Recent Files** list in the **File** menu.

---

3 Click the relevant folder in the component tree. To expand the tree and view the components, double-click closed folders. To collapse the tree, double-click open folders.

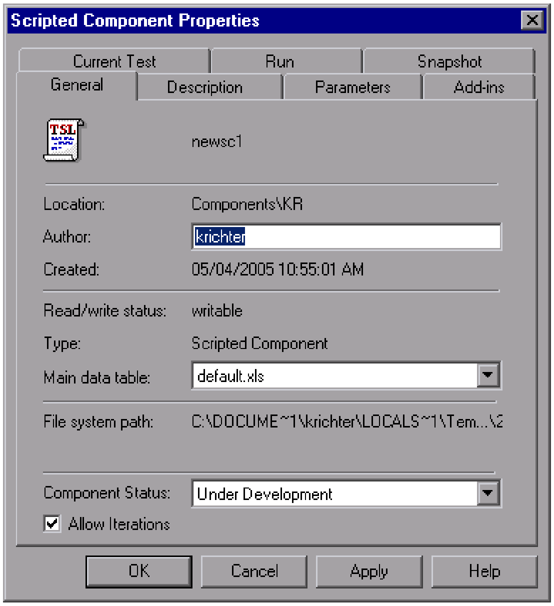
4 Select a component. The component name is displayed in the read-only **Component Name** box.

5 Click **OK** to open the component.

When the component opens, the WinRunner title bar displays the full Quality Center path and the component name. For example, the title bar for the login component may be:

Components\Flights\login

6 In the **File** menu, choose **Scripted Component Properties**. The Scripted Component Properties dialog box opens. It is divided by subject into seven tabbed pages.



newsc1

Location: Components\KR

Author: krichter

Created: 05/04/2005 10:55:01 AM

Read/write status: writable

Type: Scripted Component

Main data table: default.xls

File system path: C:\DOCUME~1\krichter\LOCALS~1\Tem...\2

Component Status: Under Development

☑ Allow Iterations

Note: The **Scripted Component Properties** command is available in the File menu only when you are connected to Quality Center with Business Process Testing.

7 To set the properties for your scripted component, select the appropriate tab and set the options, as described in the sections that follow.

8 To apply your changes and keep the Scripted Component Properties dialog box open, click **Apply**.

9 When you are finished, click **OK** and close the dialog box.

10 Close the component.

Note: Changes to parameters are saved to Quality Center only when the component is closed in WinRunner. All other changes to the Scripted Component Properties are saved when you click **Apply** or when you close the Scripted Component Properties dialog box.

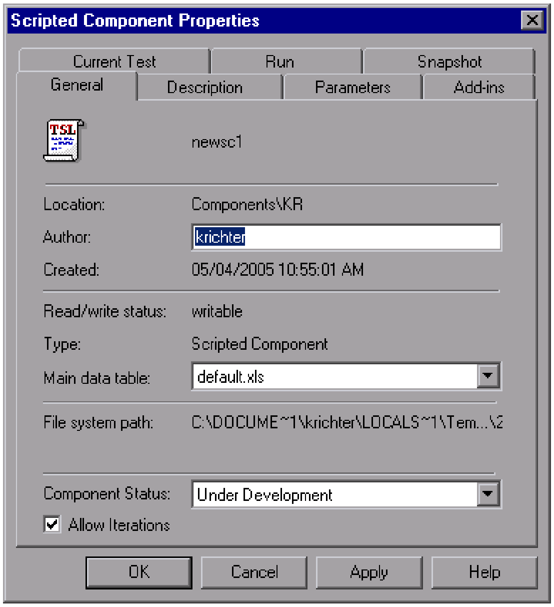
The Scripted Component Properties dialog box contains the following tabs that have special options or significance for scripted components:

Tab Heading	Description
General	Enables you to set general details about the scripted component.
Description	Enables you to enter a textual description of the scripted component.
Parameters	Enables you to define input and output parameters for the scripted component.
Snapshot	Enables you to attach an appropriate snapshot for the scripted component.

The Add-ins, Current Test and Run tabs are equally relevant for tests and components. For more information on these tabs, refer to “Setting Test Properties from the Test Properties Dialog Box” on page 510.

Defining General Details

You can document and view general details about a scripted component in the General tab of the Scripted Component Properties dialog box. You can modify the name of the author, define the component status, and choose whether the scripted component can be iterated in business process tests.



newsc1

Location: Components\KR

Author: krichter

Created: 05/04/2005 10:55:01 AM

Read/write status: writable

Type: Scripted Component

Main data table: default.xls

File system path: C:\DOCUME~1\krichter\LOCALS~1\Tem...\2

Component Status: Under Development

☑ Allow Iterations

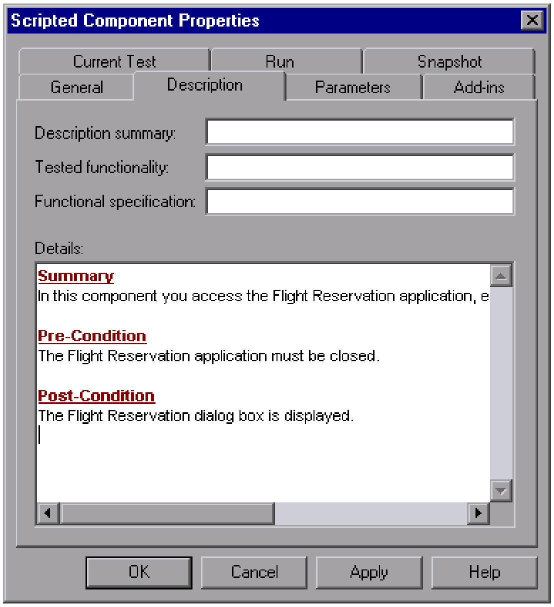
The General tab contains the following information relevant to a scripted component:

Option	Description
TSL	Displays the name of the scripted component.
Location	Displays the scripted component’s location in the Quality Center component tree.

Option	Description
Author	Enables you to enter or modify the scripted component author's name.
Created	Displays the date and time that the scripted component was created.
Read/write status	Indicates whether the scripted component is writable.
Type	Indicates that the component type is a Scripted Component.
Main data table	Enables you to select the Data Table file, either **default.xls** or a Quality Center path to a Data Table stored in Quality Center.
File system path	Displays the temporary path in the file system where the component is stored while it is open in WinRunner. This path is the cache path for Quality Center.
Component Status	Enables you to define the component's state of readiness. The status can be set to **Under Development**, **Ready**, **Maintenance** or **Error**. The status can also be set in the component's Details tab in Quality Center. For more information about component statuses, refer to the chapter, *"Getting Started with the Business Components Module"* in the *Business Process Testing User's Guide*.
Allow Iterations	Enables you to define whether multiple iterations can be set for the scripted component in a business process test. This option can also be set in the component's Details tab in Quality Center.

Defining a Component Description

The Description tab of the Scripted Component Properties dialog box enables you to view or enter a description of the scripted component.



Summary
In this component you access the Flight Reservation application, e

Pre-Condition
The Flight Reservation application must be closed.

Post-Condition
The Flight Reservation dialog box is displayed.

The Description tab contains the following information relevant to a scripted component:

Option	Description
Description summary	Enables you to specify a short summary of the component.
Tested functionality	Enables you to specify a description of the application functionality you are testing.

Option	Description
Functional specification	Enables you to specify a reference to the application's functional specification(s) for the features you are testing.
Details	Displays a textual description of the scripted component, including its pre- and post-conditions. You can enter or modify the information in this area if required.

Defining Component Parameters

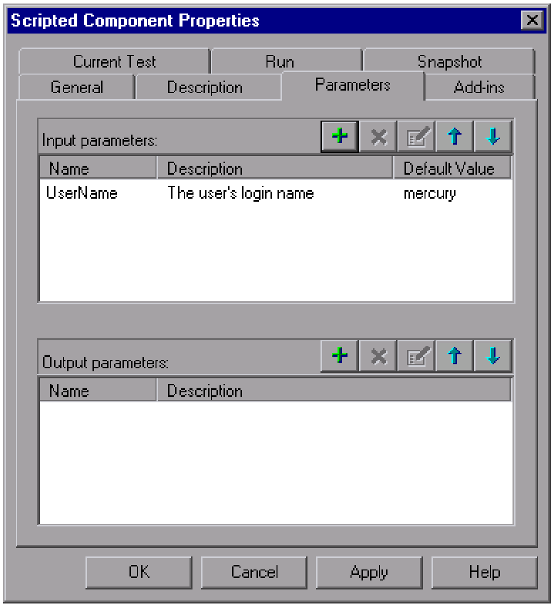
Using output and input component parameters, you can transfer data from one component to a later component in a business process test.

A *component parameter* is an element within a business process test or component that can be assigned various values. Input and output component parameters allow components to use variable values that pass values between components in the business process test. The values supplied for component parameters can affect the test results. This process greatly increases the power and flexibility of your components and business process tests.

You can define, edit, and delete parameters for the scripted component in the **Parameters** tab of the Scripted Component Properties dialog box.

The Parameters tab of the Scripted Component Properties dialog box enables you to define input component parameters that pass values into your component and output component parameters that pass values from your component to other components.

You can also use the Parameters tab to modify or delete existing component parameters.



Name	Description	Default Value
UserName	The user's login name	mercury

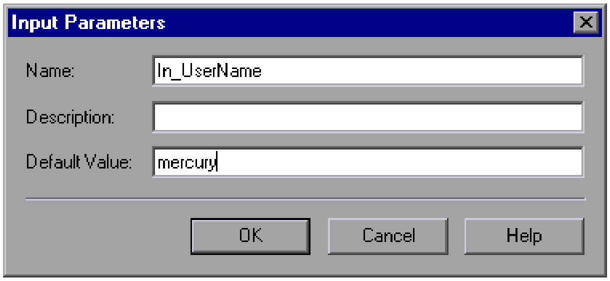
Name	Description

The Parameters tab displays the details of existing parameters for the scripted component. For more information about component parameters, refer to the *Business Process Testing User's Guide*.

Note: Changes to parameters are saved to Quality Center only when the component is closed in WinRunner. All other changes to the Scripted Component Properties are saved when you click **Apply** or when you close the Scripted Component Properties dialog box.

To define a new input or output parameter:

1 In the **Parameters** tab of the Scripted Components Properties dialog box, click the **Add** button corresponding to the parameter list (**Input** or **Output**) to which you want to add a parameter. The Input Parameters or Output Parameters dialog box opens.



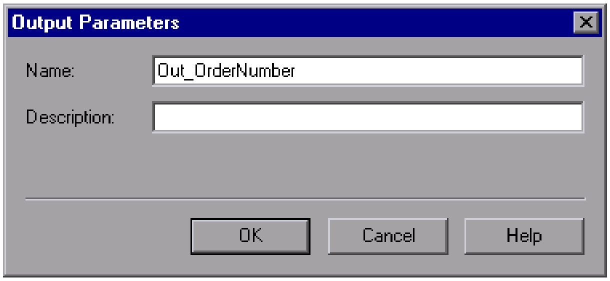
Input Parameters

Name: In_UserName

Description:

Default Value: mercury

For input parameters, the dialog box includes a text box to enter a **Default Value**. For output parameters, there is no **Default Value** edit box in the dialog box.



Output Parameters

Name: Out_OrderNumber

Description:

2 Enter a **Name** and a **Description** for the parameter. For input parameters, you can specify a **Default Value** for the parameter. The default value is used when the component runs if no other value is supplied by the business process test.

Tip: It is recommended to use IN or OUT prefixes or suffixes for the parameter names to indicate the parameter type. This makes the component steps more readable.

3 Click **OK**. The parameter is added to the appropriate parameters list.

4 If required, use the **Move Item Up** and **Move Item Down** arrow buttons to change the order of the parameters.

Note: Because parameter values are assigned sequentially, the order in which parameters are listed in the Parameters tab determines the value that is assigned to a parameter when the component is iterated.

In addition, changing the parameter order in a component that is used by a business process test could cause that test to fail.

---

5 Click **OK** to close the dialog box. The parameter details are displayed in the Parameters tab.

6 Close the component to save the parameters you have defined.

To delete a parameter from the parameter list:

1 In the **Parameters** tab of the Scripted Component Properties dialog box, select the name of the parameter to delete.

2 Click the **Delete** button corresponding to the parameter you want to delete.

3 Click **OK** to close the dialog box.

To modify a parameter in the parameter list:

1 In the **Parameters** tab of the Scripted Component Properties dialog box, select the name of the parameter to modify.

2 Click the **Modify Parameter** button or double-click the parameter name. The Input Parameters or Output Parameters dialog box opens with the current name, description (and default value if applicable), of the parameter.

3 Modify the parameter as needed.

4 Click **OK** to close the dialog box. The modified parameter is displayed in the parameters list.

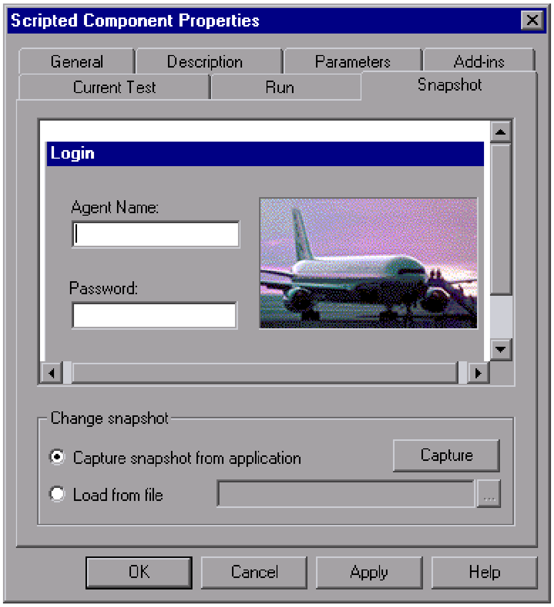
5 Close the component to save the changes you have made.

Attaching a Snapshot

You can attach an image associated with a scripted component using the **Snapshot** tab of the Scripted Component Properties dialog box.

In Quality Center, this image is displayed in business process tests that use the component. The image provides a visual indication to the Subject Matter Expert of the main purpose of the component. The image for each component in a business process test can be viewed in the Test Script tab of the Test Plan module by clicking the relevant thumbnail image. This can help the SME to quickly understand the flow of the business process test by viewing the serial flow of these images.

The Snapshot tab enables you to capture a snapshot or load a previously saved **.png** file containing an image.



Note: The snapshot image can also be captured and saved with the scripted component when working in Quality Center.

To capture and attach a snapshot:

1 In the Snapshot tab of the Scripted Component Properties dialog box, select **Capture snapshot from application** and click **Capture**. The cursor changes to a crosshairs pointer.

2 Drag the pointer to select the area in the application that you want to capture and click the right mouse button. The captured image is saved and displayed in the Snapshot tab.

3 Click **OK**.

To load an existing snapshot:

1 In the Snapshot tab of the Scripted Component Properties dialog box, select **Load from file** and click the browse button.

2 The Choose Picture File dialog box opens.

3 Browse to the required **.png** file and click **Open**. The captured image is saved and displayed in the Snapshot tab of the component.

4 Click **OK**.

Saving a Scripted Component

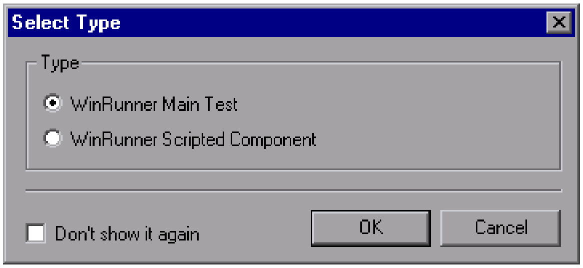
You create new components by saving an existing or new test as a scripted component. After you modify a component or test, you can save the updated component to your Quality Center project. When you save a component, you give it a descriptive name and save it to the relevant folder in the component tree in the Quality Center project (Business Components module).

You can also save a copy of an existing component to any folder in the same Quality Center project. To enable all users to differentiate between the various components, you may want to rename a copy of a component, even if you save it to a different folder.

To save a component to your Quality Center project:

1 Save the component in one of the following ways:

- ➤ To save an existing WinRunner test as a component, choose **File** > **Save as Scripted Component**.
- ➤ To save a modified existing component, click **Save**.
- ➤ To save changes to an existing component or to save a copy of an existing component, choose **File** > **Save as Scripted Component**. Proceed to step 2.
- ➤ To save an untitled test as a scripted component, click **Save**, choose **File** > **Save**, or press CTRL+S. If the component has never been saved, the Select Type dialog box opens.



Select Type

Type

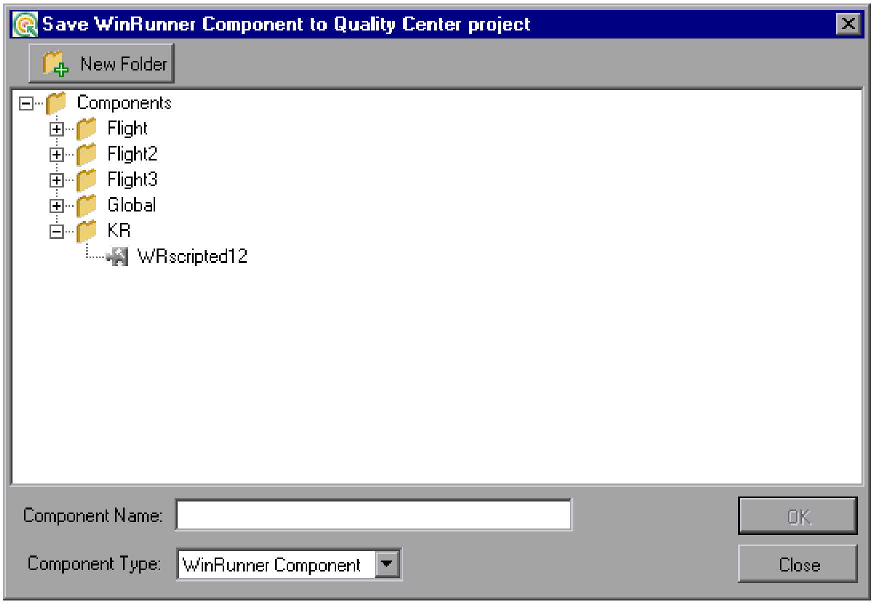
● WinRunner Main Test

○ WinRunner Scripted Component

□ Don't show it again

Select **WinRunner Scripted Component** and click **OK**.

2 The Save WinRunner Component to Quality Center Project dialog box opens and displays the component tree.



Component Type: WinRunner Component

Select the folder in which you want to save the component. To expand the tree and view a sublevel, double-click a closed folder. To collapse a sublevel, double-click an open folder.

You can either save the component to an existing folder in your Quality Center project or click the **New Folder** button to create a new folder in which to save it. If you want to save a copy of an existing component with the same name, you must save it to a different folder.

Note: Component folder names cannot include any of the following characters: \ ^ *

3 In the **Component Name** box, enter a name for the component. Use a descriptive name that will help you and others identify the component easily.

Note: Scripted component names cannot begin or end with a space, or include any of the following characters:

! @ # $ % ^ & * ( ) - + = { } [ ] | \ " ' : ; ? / < > . , ~ ‘

---

4 Accept the **Component Type—WinRunner Component**.

5 Click **OK** to save the component and close the dialog box. As WinRunner saves the component, the operations that it performs are displayed in the status bar.

The component is saved to the Quality Center project. You can now view and modify it using WinRunner.

Saving a Test as a Scripted Component

You can save an existing test as a scripted component in Quality Center with Business Process Testing. This enables a Subject Matter Expert (SME) to include the scripted component in one or more business process tests.

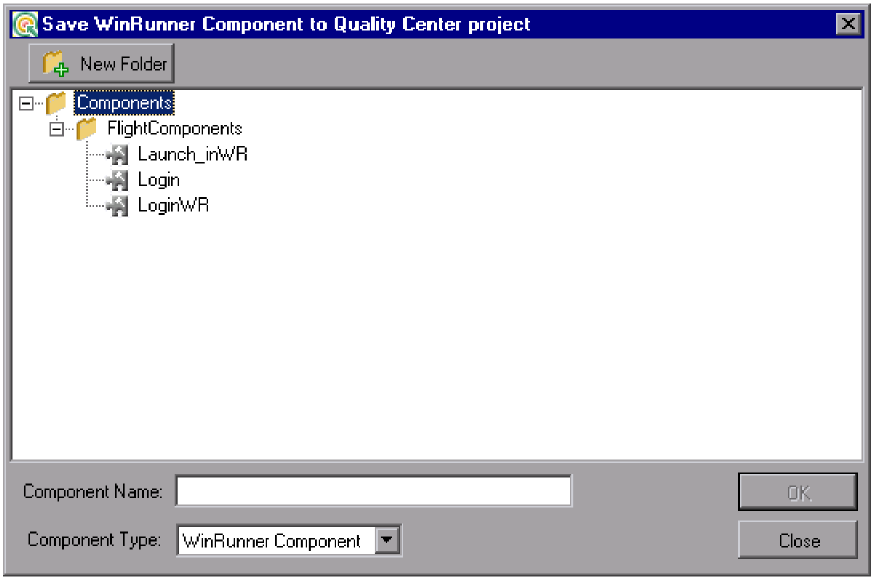
You can save a scripted component to a Quality Center database only if you are connected to a Quality Center project.

To save an existing test as a scripted component:

1 Ensure you are connected to a Quality Center project. For more information, see “Connecting to your Quality Center Project” on page 348.

2 Open a new or existing WinRunner test.

3 Choose **File** > **Save As Scripted Component**. The Save WinRunner Component to Quality Center Project dialog box opens.



The component tree from the Quality Center Business Components module is displayed.

Note: You can use the **Save as Scripted Component** command in the File menu only when you are connected to Quality Center with Business Process Testing.

---

4 Select the relevant folder in the component tree or click the **New Folder** button to create a new folder. To expand the component tree, double-click a closed folder icon. To collapse a sublevel, double-click an open folder icon.

5 In the **Component Name** text box, enter a name for the scripted component. Use a descriptive name that will help you and the SME using Quality Center to easily identify the component.

6 Click **OK** to save the component and close the dialog box.

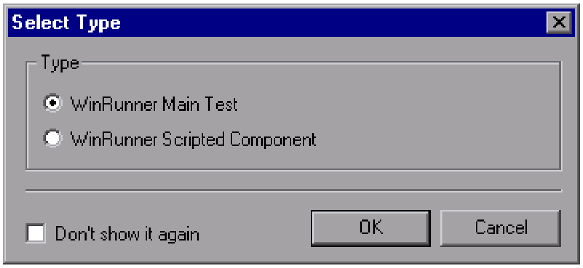
The next time you start Quality Center, or refresh the component tree in the Business Components module, the new scripted component will be displayed in the tree. Refer to the *Business Process Testing User's Guide* for more information. For more information on saving tests to a Quality Center project, see Chapter 26, "Managing the Testing Process."

Setting WinRunner Scripted Component as the Default Save Type

By default, the **Save** command and toolbar button save untitled documents as tests. You can change this default so that the **Save** command and toolbar button save untitled documents as WinRunner scripted components.

To set Scripted Component as the default save type:

1 Save a new script as a scripted component. The Select Type dialog box opens:



Select Type

Type

● WinRunner Main Test

○ WinRunner Scripted Component

□ Don't show it again

2 Select **WinRunner Scripted Component** and then select the **Don't show it again** check box.

When you next save a new script, clicking the **Save** command or toolbar button will open the Save WinRunner Component to Quality Center Project dialog box.

Tip: When you select the **Don't show it again** check box, the Select Type dialog box is not shown in the future when saving a new scripted component or test, and the toolbar **Save** button and menu command always save as the type you selected in the dialog.

If, after selecting the **Don't show it again** check box, you want to display the Select Type dialog box again, choose **Tools** > **General Options** and select the **Show Save Type dialog** checkbox.

You can also use **File** > **Save As Test** or **File** > **Save As Scripted Component** to save your component or test.

Modifying a Scripted Component

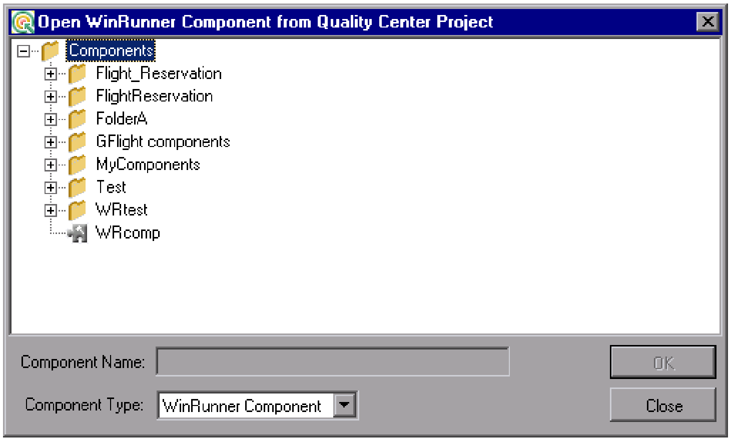
When WinRunner is connected to a Quality Center project, you can open a scripted component that is stored in the project to view, modify, debug, or run it. You find components according to their location in the component tree.

Note: Components that are currently open in Quality Center or another WinRunner session are locked and can be opened in read-only format only.

To modify a scripted component:

1 Connect to the Quality Center project in which the scripted component is stored.

2 Choose **File** > **Open Scripted Component** or press CTRL+H. The Open WinRunner Component from Quality Center Project dialog box opens.



3 Click the relevant folder in the component tree.

4 Select a **Component Type**:

- **WinRunner Component**—Displays components in the selected folder that have already been saved as WinRunner scripted components.
- **Non-automated Component**—Displays components in the selected folder that were created in Quality Center, but have not yet been automated in a Mercury testing tool. When you open a non-automated component in WinRunner, you permanently convert it to a WinRunner component.
- **All Components**—Displays all WinRunner and non-automated components in the selected folder.

5 Select a component. The component name is displayed in the read-only **Component Name** box.

6 Click **OK** to open the component. If you selected a non-automated component, it is converted to a WinRunner component.

25

Integrating with QuickTest Professional

You can design tests in QuickTest Professional and then leverage your investments in existing WinRunner script libraries by calling WinRunner tests and functions from your QuickTest test. You can also call QuickTest tests from WinRunner.

This chapter describes calling QuickTest tests from WinRunner. For information on calling WinRunner tests and functions from QuickTest, refer to the *QuickTest Professional User's Guide*.

This chapter describes:

➤ About Integrating with QuickTest Professional

➤ Calling QuickTest Tests

➤ Viewing the Results of a Called QuickTest Test

About Integrating with QuickTest Professional

If you have QuickTest Professional 6.0 or later installed on your computer, you can include calls to QuickTest tests from your WinRunner test. If you have QuickTest Professional 6.5 or later, you can call QuickTest tests and view detailed results of the test call.

You can view the detailed results of the QuickTest test run in the Unified report view of the WinRunner Test Results Window.

When WinRunner runs a called QuickTest test, it automatically loads the QuickTest add-ins required for the test, according to the associated add-ins list specified in the **Properties** tab of the QuickTest Test Settings dialog box.

Note: When using a version of QuickTest earlier than 8.0, you cannot call QuickTest tests that use QuickTest's Web Add-in from a WinRunner test if the WebTest Add-in is loaded.

For more information on working with QuickTest Add-ins, refer to the *QuickTest Professional User's Guide*.

When WinRunner is connected to a Quality Center project that contains QuickTest tests, you can call a QuickTest test that is stored in that Quality Center project.

For information on creating QuickTest tests, refer to the QuickTest Professional documentation.

Calling QuickTest Tests

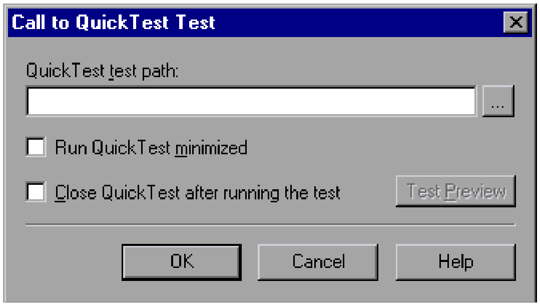
When WinRunner links to QuickTest to run a test, it starts QuickTest, opens the test (in minimized or displayed mode), and runs it. Detailed information about the results of the QuickTest test run are displayed in the Unified report view of the WinRunner Test Results window.

You can insert a call to a QuickTest test using the Call to QuickTest Test dialog box or by manually entering a **call_ex** statement.

Note: You cannot call a QuickTest test that includes calls to WinRunner tests.

To insert a call to a QuickTest test using the Call to QuickTest Test dialog box:

1 Choose **Insert** > **Call to QuickTest Test**. The Call to QuickTest Test dialog box opens.



2 In the **QuickTest test path** box, enter the path of the QuickTest test or browse to it.

If you are connected to Quality Center when you click the browse button, the Open from Quality Center project dialog box opens so that you can select the test from the Quality Center project. For more information on this dialog box, see Chapter 26, "Managing the Testing Process."

3 Select **Run QuickTest minimized** if you do not want to view the QuickTest window while the test runs. (This option is supported only for QuickTest 6.5 or later.)

4 Select **Close QuickTest after running the test** if you want the QuickTest application to close when the step calling the QuickTest test is complete.

5 Click **OK** to close the dialog box. A **call_ex** statement similar to the following is inserted in your test:

```
call_ex("F:\\Merc_Progs\\QTP\\Tests\\web\\short_flight",1,1);
```

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

call_ex (*QT_test_path* [**,** *run_minimized*, *close_QT*] **);**

Note: The **call_ex** statement provided with WinRunner 7.5 returned different values than the 7.6 version of this function. If you have tests that were created in WinRunner 7.5 and use the return value of this function, you may need to modify your test to reflect the new return values. For more information on these methods, refer to the *TSL Reference*.

For additional information on the **call_ex** function and an example of usage, refer to the *TSL Reference*.

Viewing the Results of a Called QuickTest Test

You can view the results of any WinRunner test run in the WinRunner report view or the unified report view. However, to view detailed information about a called QuickTest test (version 6.5 or later), you must ensure that WinRunner is set to generate unified report information before you run your test, and that it is set to display the unified report when you view your test results.

To instruct WinRunner to create unified report information and display the unified report:

1 Before running your test (or before displaying the test results), choose **Tools** > **General Options**. The General Options dialog box opens.

2 Click the **Run** category.

3 To ensure that unified report information is created before a test run, select **Unified report view** or select **WinRunner report view** and the **Generate unified report information** option.

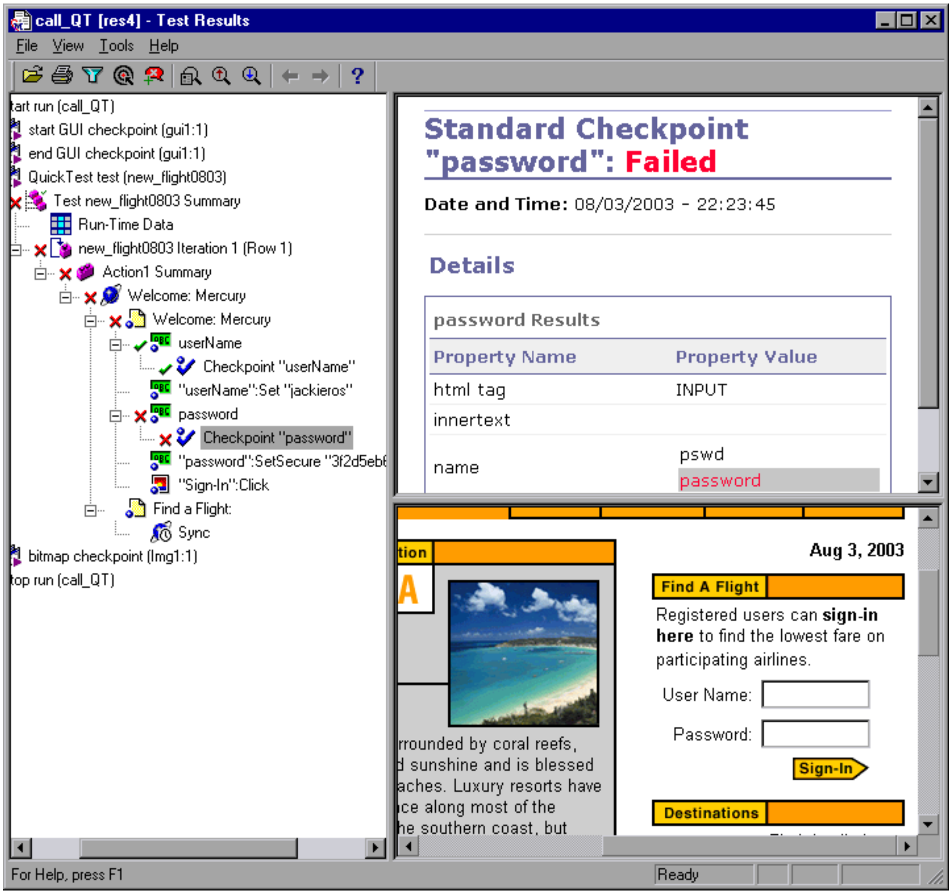
To display the unified report information, select **Unified report view** before opening the Test Results window.

For more information, refer to Chapter 21, “Analyzing Test Results” in the *Mercury WinRunner Basic Features User’s Guide*.

Analyzing the Results of a Called QuickTest Test

The unified report view of the WinRunner Test Results window includes a node for each event in your WinRunner test, plus a node for each step of the called QuickTest test.

When you select a node corresponding to a QuickTest step, the right pane displays details of the step and may contain a screen capture of the application at the time the step was performed.



```
tart run (call_QT)
start GUI checkpoint (gui1:1)
end GUI checkpoint (gui1:1)
QuickTest test (new_flight0803)
  Test new_flight0803 Summary
    Run-Time Data
    new_flight0803 Iteration 1 (Row 1)
      Action1 Summary
        Welcome: Mercury
          Welcome: Mercury
            userName
              Checkpoint "userName"
            "userName":Set "jackieros"
            password
              Checkpoint "password"
            "password":SetSecure "3f2d5eb
            "Sign-In":Click
          Find a Flight:
            Sync
bitmap checkpoint (Img1:1)
top run (call_QT)
```

Standard Checkpoint "password": Failed

Date and Time: 08/03/2003 - 22:23:45

Details

password Results

Property Name	Property Value
html tag	INPUT
innertext	
name	pswd password

tion

Aug 3, 2003

A

Find A Flight

Registered users can **sign-in here** to find the lowest fare on participating airlines.

User Name:

Password:

rrounded by coral reefs,
d sunshine and is blessed
aches. Luxury resorts have
ce along most of the
he southern coast, but

Note: You can view the results of the called QuickTest test only in the WinRunner unified report view from the results folder of the WinRunner test. The results of the QuickTest test are not saved under the called QuickTest test folder.

When analyzing the results of a WinRunner test containing a call to a QuickTest test, you may want to view the following:

- Select the **start run** node to view summary results of the WinRunner test. This summary indicates the status of the test run, but includes summary checkpoint information only for the WinRunner steps in your test.
- Select a **WinRunner** node to view the results of WinRunner events, just as you would with any WinRunner test.
- Select the QuickTest **Test** node to view summary results of the called QuickTest test. This summary includes the status of the QuickTest test run, and statistical information about the checkpoints contained in the QuickTest test.
- Select the QuickTest **Run-Time Data** node to view the resulting Data Table of the QuickTest test, including data used in Data Table parameters and data stored in the table during the test run by output values in the test.
- Select an **iteration** node to view summary information for a test iteration.
- Select an **action** node to view summary information for an action.
- Select a QuickTest step node to view information about the results of the selected step. If a screen was captured for the selected step, the captured screen is displayed in the bottom right pane of the Test Results window.

  By default, QuickTest only captures screens for failed steps. You can change the **Save step screen capture to results** option in the **Run** tab of the QuickTest Options dialog box.

For more information on the data provided for various QuickTest test steps, refer to the *QuickTest Professional User's Guide*.

For more information on analyzing WinRunner Test Results, refer to Chapter 21, "Analyzing Test Results" in the *Mercury WinRunner Basic Features User's Guide*.

26

Managing the Testing Process

Software testing typically involves creating and running thousands of tests. Quality Center (powered by TestDirector) is the Mercury application quality management solution. You can use it together with WinRunner to help you organize and control the testing process.

This chapter describes:

About Managing the Testing Process

Quality Center is a powerful test management tool that helps you systematically control the testing process. It helps you create a framework and foundation for your testing workflow.

Quality Center helps you maintain a project of tests that cover all aspects of your application's functionality. Every test in your project is designed to fulfill a specified testing requirement of your application. To meet the goals of a project, you organize the tests in your project into unique groups. Quality Center provides an intuitive and efficient method for scheduling and running tests, collecting test results, and analyzing the results.

You can save existing WinRunner tests as scripted components, or create new scripted components that can be used in Business Process Testing. Business Process Testing is a module of Quality Center that enables Subject Matter Experts (SMEs) to design quality assurance tests for an application early in the development cycle and in a script-free environment. Business Process Testing uses a new methodology for testing, and in conjunction with WinRunner, provides numerous benefits in an improved automated testing environment.

Quality Center also features a system for tracking defects, enabling you to monitor defects closely from initial detection until resolution.

WinRunner works with TestDirector 7.x and 8.0 and Quality Center.

TestDirector versions 7.5 and later, and Quality Center provide version control support, which enables you to update and revise your automated test scripts while maintaining old versions of each test. This helps you keep track of the changes made to each test script, see what was modified from one version of a script to another, or return to a previous version of the test script. For more information on version control support, see “Managing Test Versions in WinRunner” on page 396.

Note: This chapter describes the integration of WinRunner with Quality Center. For more information on working with Quality Center, refer to the *Mercury Quality Center User's Guide*. For more information about working with scripted components, refer to the *Business Process Testing User's Guide*.

Integrating the Testing Process

Quality Center and WinRunner work together to integrate all aspects of the testing process. In WinRunner, you can create scripted components and tests and save them in your Quality Center project. The components can then be included in business process tests. After you run your test, you can view and analyze the results in Quality Center.

Quality Center stores test and defect information in a project database. You can create Quality Center projects in Microsoft Access, Oracle, Sybase, or Microsoft SQL. These projects store information related to the current testing project, such as tests, test run results, and reported defects.

In order for WinRunner to access the project, you must connect it to the Web server where Quality Center is installed.

When WinRunner is connected to Quality Center, you can save a test by associating it with the Test Plan Manager. You can schedule to run a test on local or remote hosts. Test run results are sent directly to your Quality Center project.

Note: In order for Quality Center to run WinRunner tests from a remote machine, you must enable the **Allow Quality Center to run tests remotely** option from WinRunner. By default, this option is disabled. You can enable it from the **Run** category of the General Options dialog box (**Tools** > **General Options**). For more information on setting this option, refer to Chapter 23, "Setting Global Testing Options" in the *Mercury WinRunner Basic Features User's Guide*.

Accessing WinRunner Tests from Quality Center

When Quality Center accesses a WinRunner test, the test is downloaded from a project database to a local temporary directory, which becomes your current working directory. If the test calls another file (for example, a module or a test), and the full pathname of the called file is not specified, the current working directory becomes the relative path of the referenced file. Therefore, WinRunner cannot open the called test.

For example, suppose a test calls the flt_lib file:

```
static lib_path = getvar("testname") & "\\..\\flt_lib";
reload(lib_path);
```

WinRunner looks for the called test in the relative path. To enable WinRunner to find the correct pathname, you can:

- change the pathname of the WinRunner called file, or
- set direct file access for all WinRunner tests (LAN only)

Changing the Pathname of Files

To enable WinRunner to access a called file from a test, save the file in your Quality Center project and then change the pathname in your WinRunner test script.

For example, suppose you save the flt_lib file in your Quality Center project under subject\\module. Quality Center now calls the file using the following statement:

```
static lib_path = "[QC]\\Subject\\module\\flt_lib";
```

For more information on saving tests to a Quality Center project, see "Saving Tests to a Project" on page 389.

Accessing WinRunner Tests Directly (LAN only)

If you are working in a local area network (LAN) environment, you can set your machine so that it provides direct file access to all WinRunner tests, regardless of their directory path. This enables you to run WinRunner tests from Quality Center without changing the directory path of other called tests.

To set the direct file access option:

1. On the machine where WinRunner is installed, click **Run** on the **Start** menu. The Run dialog box opens.
2. Type regedit and click **OK**. The Registry Editor opens.
3. Locate the following folder:

   My Computer > **HKEY_LOCAL_MACHINE** > **Software** > **Mercury Interactive** > **TestDirector** > **Testing Tools** > **WinRunner**.
4. In the **WinRunner** folder, double-click **DirectFileAccess**. Change the value in the Value Data box to "Y".

Tip: After setting the direct access option, your Web access performance will improve while accessing WinRunner tests from Quality Center.

Connecting to and Disconnecting from a Project

If you are working with both WinRunner and Quality Center, WinRunner can communicate with your Quality Center project. You can connect or disconnect WinRunner from a Quality Center project at any time during the testing process. However, do not disconnect WinRunner from Quality Center while running tests in WinRunner from Quality Center.

The connection process has two stages. First, you connect WinRunner to the Quality Center server. This server handles the connections between WinRunner and the Quality Center project. Next, you choose the project you want WinRunner to access. The project stores the components, tests and test run information for the application you are testing. Note that Quality Center projects are password protected, so you must provide a user name and a password.

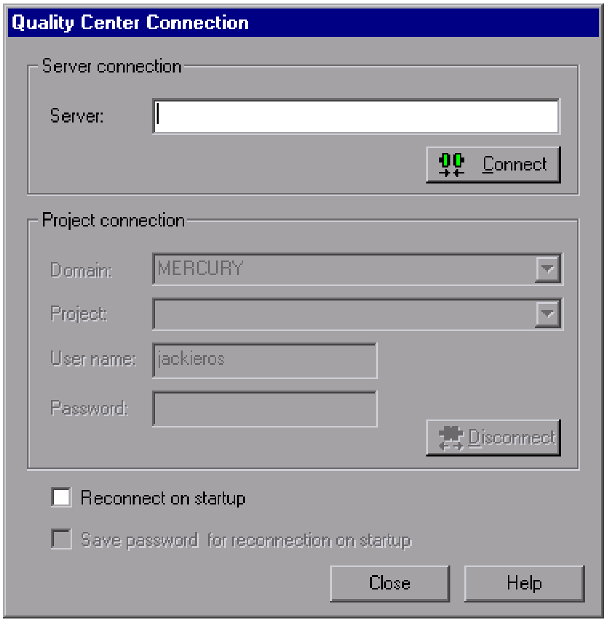
Connecting WinRunner to Quality Center

You must connect WinRunner to the server before you connect WinRunner to a project. For more information, see "Integrating the Testing Process" on page 381.

To connect WinRunner to Quality Center:

1 Choose **Tools** > **Quality Center Connection**.

The Quality Center Connection dialog box opens.



Quality Center Connection

Server connection

Server:

Project connection

Domain: MERCURY

Project:

User name: jackieros

Password:

□ Reconnect on startup

□ Save password for reconnection on startup

2 In the **Server** box, type the URL of the Web server where Quality Center is installed.

Note: You can choose a Web server accessible via a Local Area Network (LAN) or a Wide Area Network (WAN).

3 In the **Server connection** area, click **Connect**.

After the connection to the server is established, the server's name is displayed in read-only format in the **Server** box.

4 In the **Domain** box, enter or select the domain that contains the Quality Center project. (If you are working with a project in versions of TestDirector earlier than version 7.5, the **Domain** box is not relevant. Proceed to the next step.)

5 In the **Project** box, enter the Quality Center project name or select a project from the list.

6 In the **User name** box, type a user name for opening the selected project.

7 In the **Password** box, type a password for the selected project.

8 In the **Project connection** area, click **Connect** to connect WinRunner to the selected project.

After the connection to the selected project is established, the server and project connection details are displayed in read-only format.

To automatically reconnect to the Quality Center server and the selected project the next time you open WinRunner, select the **Reconnect on startup** check box.

If you do not select the **Reconnect on startup** check box, you will be prompted to connect to a Quality Center project the next time you try to create or open a scripted component.

If the **Reconnect on startup** check box is selected, then the **Save password for reconnection on startup** check box is enabled. To save your password for reconnection the next time you open WinRunner, select the **Save password for reconnection on startup** check box.

If you do not save your password, you will be prompted to enter it when WinRunner connects to Quality Center on startup.

Note: If **Reconnect on startup** is selected, but you want to open WinRunner without connecting to Quality Center, you can use the *-dont_connect* command line option as described in Chapter 15, "Running Tests from the Command Line".

9 Click **Close** to close the Quality Center Connection dialog box.

Note: You can also connect WinRunner to a Quality Center server and project using the corresponding *-qc_connection*, *-qc_database_name*, -*qc_password*, -*qc_server_name*, -*qc_user_name* command line options. For more information on these options, see "Command Line Options for Working with Quality Center" on page 410. For more information on using command line options, see Chapter 15, "Running Tests from the Command Line."

Disconnecting from Quality Center

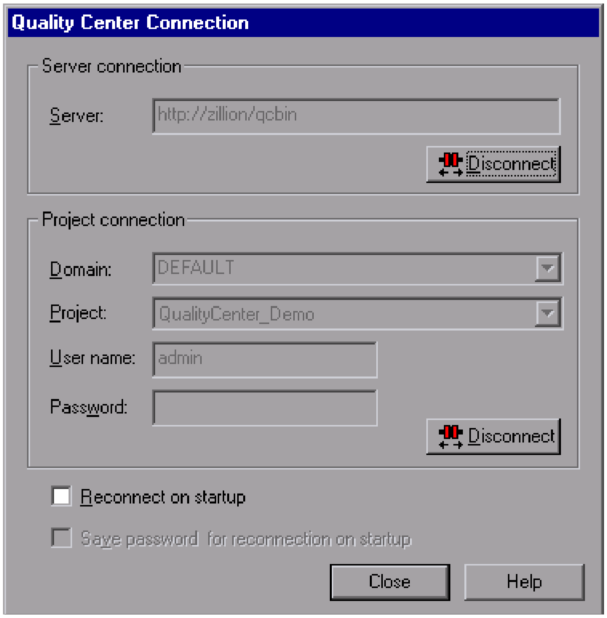
You can disconnect from a Quality Center project or server. Note that if you disconnect WinRunner from a server without first disconnecting from a project, WinRunner's connection to that database is automatically disconnected.

Note: If a test or scripted component is open when disconnecting from Quality Center, then WinRunner closes it.

To disconnect WinRunner from Quality Center:

1 Choose **Tools** > **Quality Center Connection**.

The Quality Center Connection dialog box opens.



Quality Center Connection

Server connection

Server: http://zillion/qcbin

Project connection

Domain: DEFAULT

Project: QualityCenter_Demo

User name: admin

Password:

□ Reconnect on startup

□ Save password for reconnection on startup

2 In the **Project connection** area, click **Disconnect** to disconnect WinRunner from the selected project. If you want to open a different project while using the same server, select the project as described in step 5 on page 386.

3 To disconnect WinRunner from the Quality Center server, click **Disconnect** in the **Server connection** area.

4 Click **Close** to close the Quality Center Connection dialog box.

Saving Tests to a Project

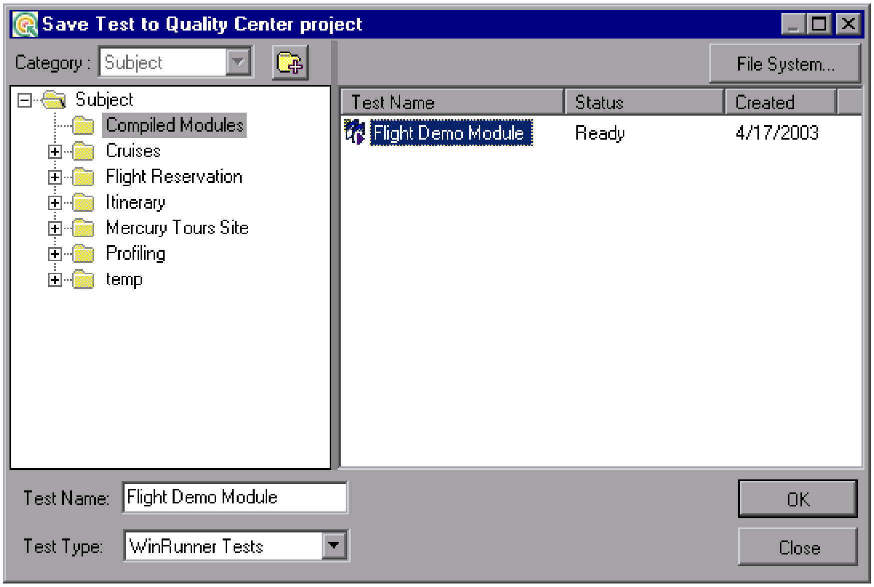
When WinRunner is connected to a Quality Center project, you can create new tests in WinRunner and save them directly to your project. To save a test, you give it a descriptive name and associate it with the relevant subject in the test plan tree. This helps you to keep track of the tests created for each subject and to quickly view the progress of test planning and creation.

Note: You can also save a test as a scripted component in Quality Center. For more information, see “Saving a Test to a Project as a Scripted Component” on page 391.

To save a test to a Quality Center project:

1 Choose **File** > **Save as Test** or click the **Save** button. For a test already saved in the file system, choose **File** > **Save As**.

If WinRunner is connected to a Quality Center project, the Save Test to Quality Center Project dialog box opens and displays the test plan tree.



Save Test to Quality Center project

Category : Subject

Test Name	Status	Created
Flight Demo Module	Ready	4/17/2003

Test Name: Flight Demo Module

Test Type: WinRunner Tests

Note that the Save Test to Quality Center Project dialog box opens only when WinRunner is connected to a Quality Center project.

Note: To save a test directly in the file system, click the **File System** button, which opens the Save Test dialog box. (From the Save Test dialog box, you may return to the Save Test to Quality Center Project dialog box by clicking the Quality Center button.)

If you save a test directly in the file system, your test will not be saved in the Quality Center project.

---

2 Select the relevant subject in the test plan tree. To expand the tree and view a sublevel, double-click a closed folder. To collapse a sublevel, double-click an open folder.

3 In the **Test Name** box, enter a name for the test. Use a descriptive name that will help you easily identify the test.

4 Click **OK** to save the test and close the dialog box.

Note: To save a batch test, choose **WinRunner Batch Tests** in the **Test Type** box.

The next time you start Quality Center, the new test will appear in the Quality Center's test plan tree. Refer to the *Mercury Quality Center User's Guide* for more information.

Saving a Test to a Project as a Scripted Component

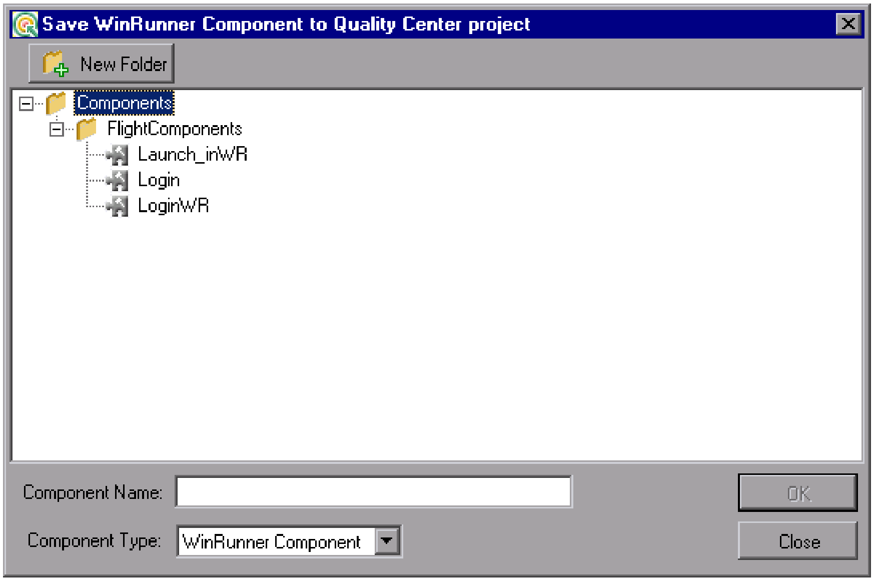
If you are working with Quality Center, you can save a new or existing test created in WinRunner directly to your project as a scripted component. The component can then be included in one or more business process tests within Quality Center. You can also define general details for the component, input and output parameters, and attach a snapshot.

Saving a Test as a Scripted Component

You can save a WinRunner test as a scripted component in the Business Components module of Quality Center.

To save a test to a Quality Center project as a scripted component:

1 After connecting to a Quality Center project, choose **File** > **Save As Scripted Component**. The Save WinRunner Component to Quality Center project dialog box opens and displays the component tree.



Note: The Save as Scripted Component option in the File menu is visible only when you are connected to Quality Center with Business Process Testing.

---

2 Select the relevant folder in the component tree or click the **New Folder** button to create a new folder. To expand the tree, double-click a closed folder icon. To collapse a sublevel, double-click an open folder icon.

3 In the **Component Name** text box, enter a name for the scripted component. Use a descriptive name that will help you easily identify the component.

4 Click **OK** to save the component and close the dialog box.

The next time you start Quality Center, or refresh the component tree in the Business Components module, the new scripted component will appear in the tree. Refer to the *Business Process Testing User's Guide* for more information.

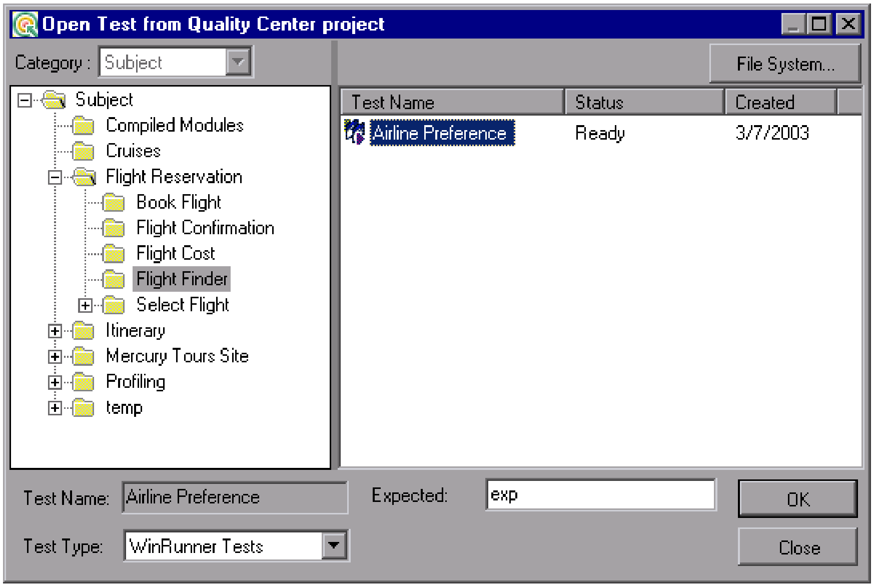
Opening Tests in a Project

If WinRunner is connected to a Quality Center project, you can open automated tests that are a part of your project. You locate tests according to their position in the test plan tree, rather than by their actual location in the file system.

To open a test saved to a Quality Center project:

1 Choose **File** > **Open Test** or click the **Open** button.

The Open Test from Quality Center Project dialog box opens and displays the test plan tree.



Test Name	Status	Created
Airline Preference	Ready	3/7/2003

Test Name: Airline Preference

Expected: exp

Test Type: WinRunner Tests

Note that the Open Test from Quality Center Project dialog box opens only when WinRunner is connected to a Quality Center project.

Note: To open a test directly from the file system, click the **File System** button, which opens the Open Test dialog box. (From the Open Test dialog box, you can return to the Open Test from Quality Center Project dialog box by clicking the Quality Center button.)

If you open a test from the file system, then when you run that test, the events of the test run will not be written to the Quality Center project.

---

2 Click the relevant subject in the test plan tree. To expand the tree and view sublevels, double-click closed folders. To collapse the tree, double-click open folders.

Note that when you select a subject, the tests that belong to the subject appear in the **Test Name** list.

3 Select a test from the **Test Name** list in the right pane. The test appears in the read-only **Test Name** box.

4 Click **OK** to open the test. The test opens in a window in WinRunner. Note that the test window's title bar shows the full subject path.

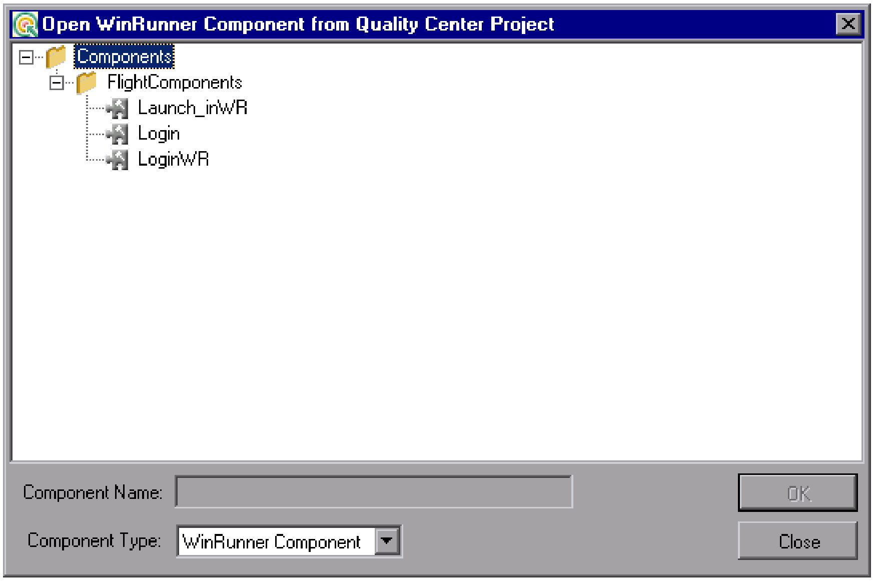
Note: To open a batch test, choose **WinRunner Batch Tests** in the **Test Type** box. For more information on batch tests, see Chapter 14, "Running Batch Tests."

Opening Scripted Components in a Project

If WinRunner is connected to a Quality Center project, you can open an existing scripted component that is a part of your project.

To open a scripted component from a Quality Center project:

1 If you are connected to a Quality Center project, choose **File** > **Open Scripted Component**. The Open WinRunner Component from Quality Center Project dialog box opens and displays the component tree.



Note: The Open Scripted Component option in the File menu is visible only when you are connected to Quality Center with Business Process Testing support.

---

2 Select the relevant component in the component tree. To expand the tree and view sublevels, double-click closed folders. To collapse the tree, double-click open folders. The scripted component appears in the read-only **Component Name** box.

3 Click **OK** to open the scripted component. The component opens in a window in WinRunner. Note that WinRunner's title bar shows the full subject path of the scripted component.

Note: The above procedure will also enable you to open a manual component, that is, a component created in Quality Center that is not yet converted to a specific testing tool type. Opening a manual component in WinRunner will set it permanently as a WinRunner component. This action cannot be reversed, even if the component is not saved in WinRunner.

Managing Test Versions in WinRunner

When WinRunner is connected to a Quality Center project with version control support, you can update and revise your automated test scripts while maintaining old versions of each test. This helps you keep track of the changes made to each test script, see what was modified from one version of a script to another, or return to a previous version of the test script.

Note: A Quality Center project with version control support requires the installation of version control software as well as Quality Center's version control software components. For more information about the Quality Center version control add-ins, refer to the *Quality Center Installation Guide*.

You manage test versions by checking tests in and out of the version control database.

Adding Tests to the Version Control Database

When you add a test to the version control database for the first time, it becomes the *Working Test* and is also assigned a permanent version number.

The working test is the test that is located in the test repository and is used by Quality Center for all test runs.

Note: Usually the latest version is the working test, but any version can be designated as the working test in Quality Center. For more information, refer to the Quality Center documentation.

To add a new test to the version control database:

1 Choose **File** > **Check In.**

Note: The Check In and Check Out options in the File menu are visible only when you are connected to a Quality Center project database with version control support, and you have a test open. The Check In option is enabled only if the active script has been saved to the project database.

2 Click **OK** to confirm adding the test to the version control database.

3 Click **OK** to reopen the checked-in test. The test will close and then reopen as a read-only file.

If you have made unsaved changes in the active test, you will be prompted to save the test.

You can review the checked-in test. You can also run the test and view the results. While the test is checked in and is in read-only format, however, you cannot make any changes to the script.

If you attempt to make changes, a WinRunner message reminds you that the script has not been checked out and that you cannot change it.

Checking Tests Out of the Version Control Database

When you open a test that is currently checked in to the version control database, you cannot make any modifications to the script. If you wish to make modifications to this script, you must check out the script.

When you check out a test, Quality Center copies the *latest version* of the test to your unique checkout directory (automatically created the first time you check out a test), and locks the test in the project database. This prevents other users of the Quality Center project from overwriting any changes you make to the test.

To check out a test:

1. Choose **File** > **Check Out**.
2. Click **OK**. The read-only test will close and automatically reopen as a writable script.

Note: The Check Out option is enabled only if the active script is currently checked in to the project's version control database.

You should check a script out of the version control database only when you want to make modifications to the script or to test the script for workability.

Checking Tests In to the Version Control Database

When you have finished making changes to a test you check it in to the version control database in order to make it the new *latest version* and to assign it as the *working test*.

When you check a test back into the version control database, Quality Center deletes the test copy from your checkout directory and unlocks the test in the database so that the test version will be available to other users of the Quality Center project.

To check in a test:

1. Choose **File** > **Check In**.
2. Click **OK**. The file will close and automatically reopen as a read-only script.

If you run tests after you have checked in the script, the results will be saved to the Quality Center Project database.

Tip: You should close a test in WinRunner before using Quality Center to change the checked in/checked out status of the test. If you make changes to the test's status using Quality Center while the test is open in WinRunner, WinRunner will not reflect those changes. For more information, refer to the Quality Center documentation.

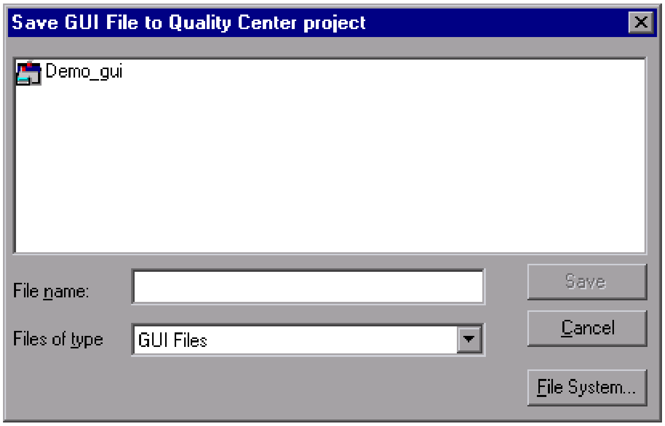
Saving GUI Map Files to a Project

When WinRunner is connected to a Quality Center project, choose **File** > **Save** in the GUI Map Editor to save your GUI map file to the open database. All the GUI map files used in all the tests saved to the Quality Center project are stored together. This facilitates keeping track of the GUI map files associated with tests in your project.

To save a GUI map file to a Quality Center project:

1. Choose **Tools** > **GUI Map Editor** to open the GUI Map Editor.
2. From a temporary GUI map file, choose **File** > **Save**. From an existing GUI map file, choose **File** > **Save As**.

The Save GUI File to Quality Center project dialog box opens. If any GUI map files have already been saved to the open database, they are listed in the dialog box.



Note that the Save GUI File to Quality Center Project dialog box opens only when WinRunner is connected to a Quality Center project.

To save a GUI map file directly to the file system, click the **File System** button, which opens the Save GUI File dialog box. (From the Save GUI File dialog box, you may return to the Save GUI File to Quality Center Project dialog box by clicking the Quality Center button.)

Note: If you save a GUI map file directly to the file system, your GUI map file will not be saved in the Quality Center project.

---

3 In the **File name** text box, enter a name for the GUI map file. Use a descriptive name that will help you easily identify the GUI map file.

4 Click **Save** to save the GUI map file and to close the dialog box.

Note: When you choose to save a GUI map file to a Quality Center project, it is uploaded to the project immediately.

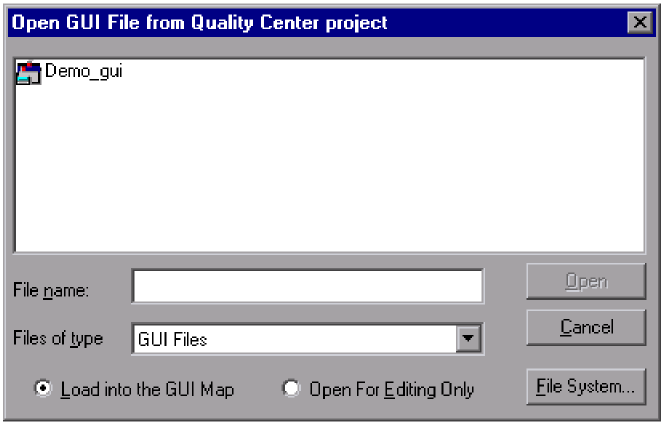
Opening GUI Map Files in a Project

When WinRunner is connected to a Quality Center project, you can use the GUI Map Editor to open a GUI map file saved to a Quality Center project.

To open a GUI map file saved to a Quality Center project:

1. Choose **Tools** > **GUI Map Editor** to open the GUI Map Editor.
2. In the GUI Map Editor, choose **File** > **Open**.

   The Open GUI File from Quality Center project dialog box opens. All the GUI map files that have been saved to the open database are listed in the dialog box.

   Open GUI File from Quality Center project

   Demo_gui

   File name:

   Files of type: GUI Files

   ● Load into the GUI Map ○ Open For Editing Only

   Note that the Open GUI File from Quality Center project dialog box opens only when WinRunner is connected to a Quality Center project.

   To open a GUI map file directly from the file system, click the **File System** button, which opens the Open GUI File dialog box. (From the Open GUI File dialog box, you may return to the Open GUI File from Quality Center Project dialog box by clicking the Quality Center button.)
3. Select a GUI map file from the list of GUI map files in the open database. The name of the GUI map file appears in the **File name** box.

4 To load the GUI map file to open into the GUI Map Editor, click **Load into the GUI Map**. Note that this is the default setting. Alternatively, if you only want to edit the GUI map file, click **Open for Editing Only**. For more information, refer to Chapter 7, "Editing the GUI Map" in the *Mercury WinRunner Basic Features User's Guide*.

5 Click **Open** to open the GUI map file. The GUI map file is added to the GUI file list. The letter "L" indicates that the file is loaded.

Running Tests in a Test Set

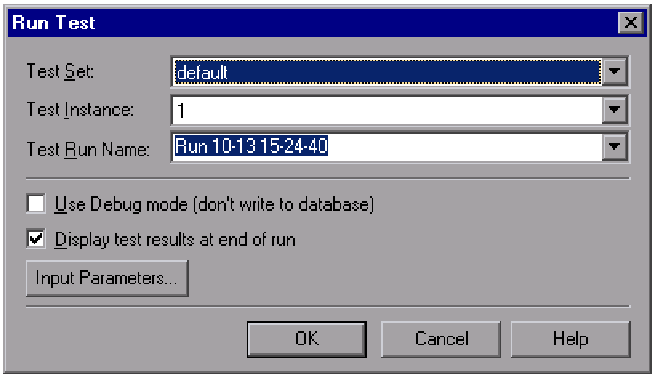
A test set is a group of tests selected to achieve specific testing goals. For example, you can create a test set that tests the user interface of the application or the application's performance under stress. You define test sets when working in Quality Center's test run mode.

If WinRunner is connected to a project and you want to run tests in the project from WinRunner, specify the name of the current test set before you begin. When the test run is completed, the tests are stored in Quality Center according to the test set you specified.

To specify a test set and user name:

1 Choose a **Run** command from the **Test** menu.

The Run Test dialog box opens.



Run Test

Test Set: default

Test Instance: 1

Test Run Name: Run 10-13 15-24-40

☐ Use Debug mode (don't write to database)

☑ Display test results at end of run

2. In the **Test Set** box, select a test set from the list. The list contains test sets created in Quality Center.

3. If your test set contains more than one instance of the test, select the **Test Instance**. If you are working with a version of TestDirector earlier than 7.5, the Test Instance is always 1.

4. In the **Test Run Name** box, select a name for this test run, or enter a new name.

5. To run tests in **Debug** mode, select the **Use Debug mode** check box. If this option is selected, the results of this test run are not written to the Quality Center project.

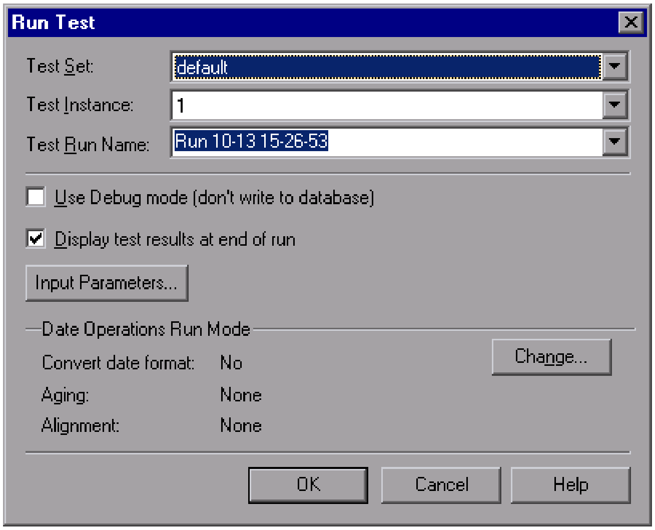
6. To display the test results in WinRunner at the end of a test run, select the **Display test results at end of run** check box.

7. To supply values for input parameters, click **Input Parameters** and enter the values you want to use for this test run in the Input Parameters dialog box. For more information, refer to Chapter 20, “Understanding Test Runs” in the *Mercury WinRunner Basic Features User’s Guide*.

8. Click **OK** to save the parameters and to run the test.

Running Date Operations Tests in a Test Set

If the **Enable date operations** option is selected (**Tools** > **General Options** > **General** category), you can also view and modify the Date Operations Run Mode settings from the Run Test dialog box.



Run Test

Test Set: default

Test Instance: 1

Test Run Name: Run 10-13 15-26-53

☐ Use Debug mode (don't write to database)

☑ Display test results at end of run

Date Operations Run Mode

Convert date format: No

Aging: None

Alignment: None

For more information on running tests to check date operations, refer to Chapter 20, “Understanding Test Runs” in the *Mercury WinRunner Basic Features User’s Guide*.

Running Tests on Remote Hosts

You can run WinRunner tests on multiple remote hosts. To enable another Mercury product to use a computer as a remote host, you must activate the **Allow other Mercury products to run tests remotely** option. Note that when you run a test on a remote host, you should run the test in silent mode, which suppresses WinRunner messages during a test run. For more information on silent mode, see Chapter 21, “Setting Testing Options from a Test Script.”

To enable another Mercury product on a remote machine to run WinRunner tests:

1. Choose **Tools** > **General Options** to open the General Options dialog box.
2. Click the **Run** category.
3. Select the **Allow other Mercury products to run tests remotely** check box.

Note: If the **Allow other Mercury Products to run tests remotely** check box is cleared, WinRunner tests can only be run locally.

For more information on setting testing options using the General Options dialog box, refer to Chapter 23, “Setting Global Testing Options” in the *Mercury WinRunner Basic Features User’s Guide*.

Viewing Test Results from a Project

If you run tests in a test set, you can view the test results from a Quality Center project. If you run a test set in **Verify** mode, the Test Results window opens automatically at the end of the test run. At other times, choose **Tools** > **Test Results** to open the Test Results window. By default, the Test Results window displays the test results of the last test run of the active test. To view the test results for another test or for an earlier test run of the active test, choose **File** > **Open** in the Test Results window.

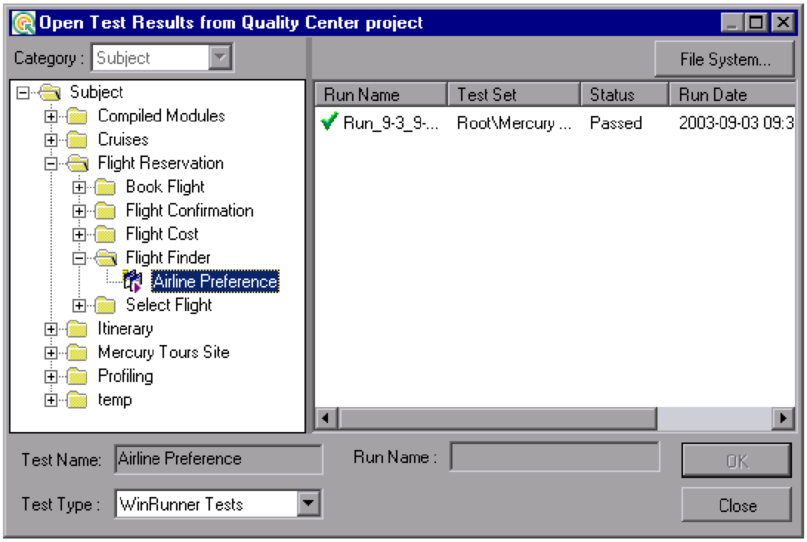
To view test results from a Quality Center project:

1. Choose **Tools** > **Test Results**.

   The Test Results window opens, displaying the test results of the last test run of the active test.

2. In the Test Results window, choose **File** > **Open**.

The Open Test Results from Quality Center project dialog box opens and displays the test plan tree.



Open Test Results from Quality Center project

Category : Subject

Run Name	Test Set	Status	Run Date
✓ Run_9-3_9-...	Root\Mercury ...	Passed	2003-09-03 09:3

Test Name: Airline Preference

Run Name :

Test Type : WinRunner Tests

Note that the Open Test Results from Quality Center project dialog box opens only when WinRunner is connected to a Quality Center project.

To open test results directly from the file system, click the **File System** button, which opens the Open Test Results dialog box. (From the Open Test Results dialog box, you may return to the Open Test Results from Quality Center Project dialog box by clicking the Quality Center button.)

3 In the **Test Type** box, select the type of test to view in the dialog box: all tests (the default setting), WinRunner tests, or WinRunner batch tests.

4 Select the relevant subject in the test plan tree. To expand the tree and view a sublevel, double-click a closed folder. To collapse a sublevel, double-click an open folder.

5 Select a test run to view. In the right pane:

- The **Run Name** column displays whether your test run passed or failed and contains the names of the test runs.
- The **Test Set** column contains the names of the test sets.
- Entries in the **Status** column indicate whether the test passed or failed.
- The **Run Date** column displays the date and time when the test set was run.

6 Click **OK** to view the results of the selected test.

If the test results indicate defects in your application, you can report the defects to your Quality Center defect database directly from the Test Results window. For more information, refer to Chapter 21, "Analyzing Test Results." in the *Mercury WinRunner Basic Features User's Guide*.

For information about the options in the Test Results window, refer to Chapter 21, "Analyzing Test Results" in the *Mercury WinRunner Basic Features User's Guide*.

Using TSL Functions with Quality Center

Several TSL functions facilitate your work with a Quality Center project by returning the values of fields in a Quality Center project. In addition, working with Quality Center facilitates working with many TSL functions: when WinRunner is connected to Quality Center, you can specify a path in a Quality Center project in a TSL statement instead of using the full file system path.

Quality Center Project Functions

Several TSL functions enable you to retrieve information from a Quality Center project.

qcdb_add_defect	Adds a new defect to the Quality Center defect database for the project to which WinRunner is connected.

qcdb_get_step_value	Returns the value of a field in the "dessteps" table in a Quality Center project.
qcdb_get_test_value	Returns the value of a field in the "test" table in a Quality Center project.
qcdb_get_testset_value	Returns the value of a field in the "testcycl" table in a Quality Center project.
qcdb_load_attachment	Downloads a file attachment of a test to the local cache and returns its location.

You can use the Function Generator to insert these functions into your test scripts, or you can manually program statements that use them.

For more information about these functions, refer to the *TSL Reference*.

Call Statements and Compiled Module Functions

When WinRunner is connected to Quality Center, you can specify the paths of tests and compiled module functions saved in a Quality Center project when you use the **call**, **call_close**, **load**, **reload**, and **unload** functions.

For example, if you have a test with the following path in your Quality Center project, Subject\Sub1\My_test, you can call it from your test script with the statement:

```
call "[QC]\\Subject\\Sub1\\My_test"();
```

Alternatively, if you specify the “[QC]\Subject\Sub1” search path in the **Folders** category of the General Options dialog box or by using a **setvar** statement in your test script, you can call the test from your test script with the following statement:

```
call "My_test" ();
```

Note that the [QC] prefix is optional when specifying a test or a compiled module in a Quality Center project.

Note: When you run a WinRunner test from a Quality Center project, you can specify its "In" parameters from within Quality Center, instead of using **call** statements to pass parameters from a test to a called test. You may not use Quality Center to call a WinRunner test that has "Out" parameters defined. For information about specifying parameters for WinRunner tests from Quality Center, refer to the *Mercury Quality Center User's Guide*. For information on "In" and "Out" parameters, see "About Calling Tests" on page 131.

For more information on working with the specified Call Statement and Compiled Module functions, refer to the *TSL Reference*.

GUI Map Editor Functions

When WinRunner is connected to Quality Center, you can specify the names of GUI map files saved in a Quality Center project when you use GUI Map Editor functions in a test script.

When WinRunner is connected to a Quality Center project, WinRunner stores GUI map files in the GUI repository in the database. Note that the [QC] prefix is optional when specifying a GUI map file in a Quality Center project.

For example, if the My_gui.gui GUI map file is stored in a Quality Center project, in My_project_database\GUI, you can load it with the statement:

```
GUI_load ("My_gui.gui");
```

For information about working with GUI Map Editor functions, refer to the *TSL Reference*.

Specifying Search Paths for Tests Called from Quality Center

You can configure WinRunner to use search paths based on the path in a Quality Center project.

In the following example, a **setvar** statement specifies a search path in a Quality Center project:

```
setvar ( "searchpath", "[QC]\\My_project_database\\Subject\\Sub1" );
```

For information on how to specify the search path using the General Options dialog box, refer to Chapter 23, “Setting Global Testing Options” in the *Mercury WinRunner Basic Features User’s Guide*. For information on how to specify the search path by using a **setvar** statement, see Chapter 21, “Setting Testing Options from a Test Script.”

Command Line Options for Working with Quality Center

You can use the Windows Run command to set parameters for working with Quality Center. You can also save your startup parameters by creating a custom WinRunner shortcut. Then, to start WinRunner with the startup parameters, you simply double-click the icon.

You can use the command line options described below to set parameters for working with Quality Center. For additional information on using command line options, see Chapter 15, “Running Tests from the Command Line.”

-dont_connect

If the **Reconnect on startup** check box is selected in the Quality Center Connection dialog box, this command line enables you to open WinRunner without connecting to Quality Center.

-qc_connection {on | off}

Activates WinRunner’s connection to Quality Center when set to **on**.

(Default = **off**)

(Formerly **-test_director.**)

Note: If you select the "Reconnect on startup" option in the Connection to Test Director dialog box, setting **-qc_connection** to off will not prevent the connection to Quality Center. To prevent the connection to Quality Center in this situation, use the **-dont_connect** command. For more information, see "-dont_connect," on page 410.

-qc_cycle_name *cycle_name*

Specifies the name of the current test cycle. This option is applicable only when WinRunner is connected to Quality Center.

Note that you can use the corresponding *qc_cycle_name* testing option to specify the name of the current test cycle, as described in Chapter 21, "Setting Testing Options from a Test Script."

-qc_database_name *database_pathname*

Specifies the active Quality Center project. WinRunner can open, execute, and save tests in this project. This option is applicable only when WinRunner is connected to Quality Center.

Note that you can use the corresponding *qc_database_name* testing option to specify the active Quality Center database, as described in Chapter 21, "Setting Testing Options from a Test Script."

Note that when WinRunner is connected to Quality Center, you can specify the active Quality Center project from the Quality Center Connection dialog box, which you open by choosing **Tools** > **Quality Center Connection**. For more information about connecting to Quality Center, see "Connecting to and Disconnecting from a Project" on page 384.

-qc_password

Specifies the password for connecting to a project in a Quality Center server.

Note that you can specify the password for connecting to Quality Center from the Quality Center Connection dialog box, which you open by choosing **Tools** > **Quality Center Connection**.

For more information about connecting to Quality Center, see "Connecting to and Disconnecting from a Project" on page 384.

-qc_server_name

Specifies the name of the Quality Center server to which WinRunner connects.

Note that you can use the corresponding *qc_server_name* testing option to specify the name of the Quality Center server to which WinRunner connects, as described in Chapter 21, "Setting Testing Options from a Test Script."

Note that you can specify the name of the Quality Center server to which WinRunner connects from the Quality Center Connection dialog box, which you open by choosing **Tools** > **Quality Center Connection**. For more information about connecting to Quality Center, see "Connecting to and Disconnecting from a Project" on page 384.

-qc_user_name ***user_name***

Specifies the name of the user who is currently executing a test cycle. (Formerly *user.*)

Note that you can use the corresponding *qc_user_name* testing option to specify the user, as described in Chapter 21, "Setting Testing Options from a Test Script."

Note that you can specify the user name when you connect to Quality Center from the Quality Center Connection dialog box, which you open by choosing **Tools** > **Quality Center Connection**. For more information about connecting to Quality Center, see “Connecting to and Disconnecting from a Project” on page 384.

For more information on using command line options, see Chapter 15, “Running Tests from the Command Line.”

27

Testing Systems Under Load

Today's applications are run by multiple users over complex architectures. With LoadRunner, the Mercury solution for automated performance testing, you can test the performance and reliability of an entire system.

This chapter describes:

- About Testing Systems Under Load
- Emulating Multiple Users
- Virtual User (Vuser) Technology
- Developing and Running Scenarios
- Creating GUI Vuser Scripts
- Measuring Server Performance
- Synchronizing Vuser Transactions
- Creating a Rendezvous Point
- A Sample Vuser Script

About Testing Systems Under Load

Software testing is no longer confined to testing applications that run on a single, standalone PC. Applications are run in network environments where multiple client PCs or UNIX workstations interact with a central server. Web-based applications are also common.

Modern architectures are complex. While they provide an unprecedented degree of power and flexibility, these systems are difficult to test. LoadRunner emulates load and then accurately measures and analyzes performance and functionality. This chapter provides an overview of how to use WinRunner together with LoadRunner to test your system. For detailed information about how to load test an application, refer to the LoadRunner documentation.

Emulating Multiple Users

With LoadRunner, you emulate the interaction of multiple users by creating *scenarios*. A scenario defines the events that occur during each load testing session, such as the number of users, the actions they perform, and the machines they use. For more information about scenarios, refer to the *LoadRunner Controller User's Guide*.

In a scenario, LoadRunner replaces the human user with a *virtual user* or *Vuser*. A Vuser emulates the actions of a human user working with your application. A scenario can contain tens, hundreds, or thousands of Vusers.

Virtual User (Vuser) Technology

LoadRunner provides a variety of Vuser technologies that enable you to generate load when using different types of system architectures. Each Vuser technology is suited to a particular architecture, and results in a specific type of Vuser. For example, you use GUI Vusers to operate graphical user interface applications in environments such as Microsoft Windows; Web Vusers to emulate users operating Web browsers; RTE Vusers to operate terminal emulators; Database Vusers to emulate database clients communicating with a database application server.

The various Vuser technologies can be used alone or together, to create effective load testing scenarios.

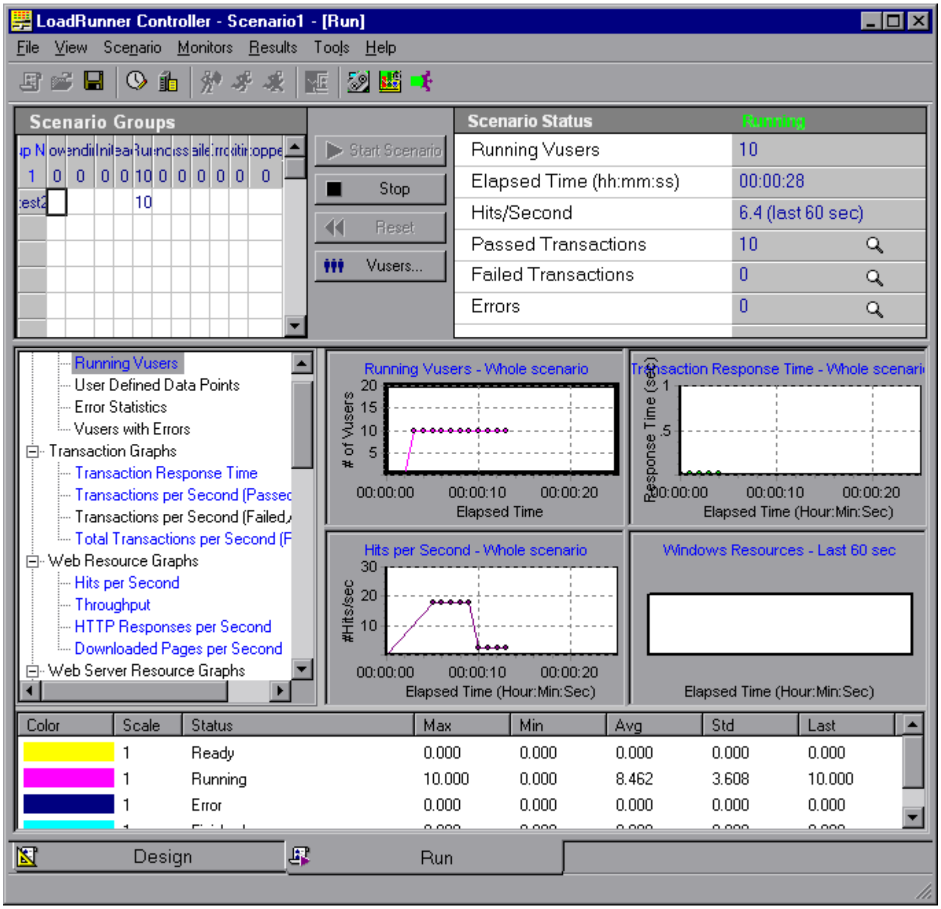
GUI Vusers

GUI Vusers operate graphical user interface applications in environments such as Microsoft Windows. Each GUI Vuser emulates a real user submitting input to and receiving output from a client application.

A GUI Vuser consists of a copy of WinRunner and a client application. The client application can be any application used to access the server, such as a database client. WinRunner replaces the human user and operates the client application. Each GUI Vuser executes a Vuser script. This is a WinRunner test that describes the actions that the Vuser will perform during the scenario. It includes statements that measure and record the performance of the server. For more information, refer to the LoadRunner *Creating Vuser Scripts* guide.

Developing and Running Scenarios

You use the LoadRunner Controller to develop and run scenarios. The Controller is an application that runs on any network PC.


LoadRunner Controller - Scenario1 - [Run]
File View Scenario Monitors Results Tools Help
Scenario Groups
Start Scenario
Stop
Reset
Vusers...
Running Vusers
User Defined Data Points
Error Statistics
Vusers with Errors
Transaction Graphs
Transaction Response Time
Transactions per Second (Passed
Transactions per Second (Failed,
Total Transactions per Second (F
Web Resource Graphs
Hits per Second
Throughput
HTTP Responses per Second
Downloaded Pages per Second
Web Server Resource Graphs
Running Vusers - Whole scenario
# of Vusers
Elapsed Time
Transaction Response Time - Whole scenario
Response Time (sec)
Elapsed Time (Hour:Min:Sec)
Hits per Second - Whole scenario
#Hits/sec
Elapsed Time (Hour:Min:Sec)
Windows Resources - Last 60 sec
Elapsed Time (Hour:Min:Sec)
00:00:00
00:00:10
00:00:20
Design
Run

Scenario Status	Running
Running Vusers	10
Elapsed Time (hh:mm:ss)	00:00:28
Hits/Second	6.4 (last 60 sec)
Passed Transactions	10
Failed Transactions	0
Errors	0

Color	Scale	Status	Max	Min	Avg	Std	Last
🟨	1	Ready	0.000	0.000	0.000	0.000	0.000
🟪	1	Running	10.000	0.000	8.462	3.608	10.000
🟦	1	Error	0.000	0.000	0.000	0.000	0.000

The following procedure outlines how to use the LoadRunner Controller to create, run, and analyze a scenario. For more information, refer to the *LoadRunner Controller User's Guide*.

1 **Invoke the Controller.**

2 **Create the scenario.**

A scenario describes the events that occur during each load testing session, such as the participating Vusers, the scripts they run, and the machines the Vusers use to run the scripts (load generating machines).

3 **Run the scenario.**

When you run the scenario, LoadRunner distributes the Vusers to their designated load generating machines. When the load generating machines are ready, they begin executing the scripts. During the scenario run, LoadRunner measures and records server performance data, and provides online network and server monitoring.

4 **Analyze server performance.**

After the scenario runs, you can use LoadRunner's graphs and reports to analyze server performance data captured during the scenario run.

The rest of this chapter describes how to create GUI Vuser scripts. These scripts describe the actions performed by a human user accessing a server from an application running on a client PC.

Creating GUI Vuser Scripts

A GUI Vuser script describes the actions a GUI Vuser performs during a LoadRunner scenario. You use WinRunner to create GUI Vuser scripts. The following procedure outlines the process of creating a basic script. For a detailed explanation, refer to the LoadRunner *Creating Vuser Scripts* guide.

To create a GUI Vuser script:

1 Start WinRunner.

2 Start the client application.

3 Record operations on the client application.

4 Edit the Vuser script using WinRunner, and program additional TSL statements. Add control-flow structures as needed.

5 Define actions within the script as transactions to measure server performance.

6 Add synchronization points to the script.

7 Add *rendezvous* points to the script to coordinate the actions of multiple Vusers.

8 Save the script and exit WinRunner.

Measuring Server Performance

Transactions measure how your server performs under the load of many users. A transaction may be a simple task, such as entering text into a text field, or it may be an entire test that includes multiple tasks. LoadRunner measures the performance of a transaction under different loads. You can measure the time it takes a single user or a hundred users to perform the same transaction.

The first stage of creating a transaction is to declare its name at the start of the Vuser script. When you assign the Vuser script to a Vuser, the Controller scans the Vuser script for transaction declaration statements. If the script contains a transaction declaration, LoadRunner reads the name of the transaction and displays it in the Transactions window.

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

declare_transaction ("*transaction_name***");**

The *transaction_name* must be a string constant, not a variable or an expression. This string can contain up to 128 characters. No spaces are permitted.

Next, mark the point where LoadRunner will start to measure the transaction. Insert a **start_transaction** statement into the Vuser script immediately before the action you want to measure.

The syntax of this function is:

start_transaction ("*transaction_name*" **);**

The *transaction_name* is the name you defined in the **declare_transaction** statement.

Insert an **end_transaction** statement into the Vuser script to indicate the end of the transaction. If the entire test is a single transaction, then insert this statement in the last line of the script. The syntax of this function is:

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

The *transaction_name* is the name you defined in the **declare_transaction** statement. The *status* tells LoadRunner to end the transaction only if the transaction passed (PASS) or failed (FAIL).

Synchronizing Vuser Transactions

For transactions to accurately measure server performance, they must reflect the time the server takes to respond to user requests. A human user knows that the server has completed processing a task when a visual cue, such as a message, appears. For instance, suppose you want to measure the time it takes for a database server to respond to user queries. You know that the server completed processing a database query when the answer to the query is displayed on the screen. In Vuser scripts, you instruct the Vusers to wait for a cue by inserting synchronization points.

Synchronization points tell the Vuser to wait for a specific event to occur, such as the appearance of a message in an object, and then resume script execution. If the object does not appear, the Vuser continues to wait until the object appears or a time limit expires. You can synchronize transactions by using any WinRunner synchronization or object function. For more information on WinRunner synchonization functions, refer to Chapter 19, “Synchronizing the Test Run” in the *Mercury WinRunner Basic Features User’s Guide*.

Creating a Rendezvous Point

During the scenario run, you instruct multiple Vusers to perform tasks simultaneously by creating a rendezvous point. This ensures that:

- intense user load is emulated
- transactions are measured under the load of multiple Vusers

A rendezvous point is a meeting place for Vusers. You designate the meeting place by inserting rendezvous statements into your Vuser scripts. When the rendezvous statement is interpreted, the Vuser is held by the Controller until all the members of the rendezvous arrive. When all the Vusers have arrived (or a time limit is reached), they are released together to perform the next task in their Vuser scripts.

The first stage of creating a rendezvous point is to declare its name at the start of the Vuser script. When you assign the Vuser script to a Vuser, LoadRunner scans the script for rendezvous declaration statements. If the script contains a rendezvous declaration, LoadRunner reads the rendezvous name and creates a rendezvous. If you create another Vuser that runs the same script, the Controller will add the Vuser to the rendezvous.

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 of the rendezvous. The *rendezvous_name* must be a string constant, not a variable or an expression. This string can contain up to 128 characters. No spaces are permitted.

Next, you indicate the point in the Vuser script where the rendezvous will occur by inserting a **rendezvous** statement. This tells LoadRunner to hold the Vuser at the rendezvous until all the other Vusers arrive. The function has the following syntax:

rendezvous ("*rendezvous_name*" **)**;

The *rendezvous_name* is the name of the rendezvous. The *rendezvous_name* must be a string constant, not a variable or an expression. This string can contain up to 128 characters. No spaces are permitted.

A Sample Vuser Script

In the following sample Vuser script, the “Ready” transaction measures how long it takes for the server to respond to a request from a user. The user enters the request and then clicks OK. The user knows that the request has been processed when the word “Ready” appears in the client application’s Status text box.

In the first part of the Vuser script, the **declare_transaction** and **declare_rendezvous** functions declare the names of the transaction and rendezvous points in the Vuser script. In this script, the transaction “Ready” and the rendezvous “wait” are declared. The declaration statements enable the LoadRunner Controller to display transaction and rendezvous information.

```
# Declare the transaction name
declare_transaction ("Ready");

# Define the rendezvous name
declare_rendezvous ("wait");
```

Next, a **rendezvous** statement ensures that all Vusers click OK at the same time, in order to create heavy load on the server.

```
# Define rendezvous points
rendezvous ("wait");
```

In the following section, a **start_transaction** statement is inserted just before the Vuser clicks OK. This instructs LoadRunner to start recording the “Ready” transaction. The “Ready” transaction measures the time it takes for the server to process the request sent by the Vuser.

```
# Deposit transaction
start_transaction ( "Ready" );
button_press ( "OK" );
```

Before LoadRunner can measure the transaction time, it must wait for a cue that the server has finished processing the request. A human user knows that the request has been processed when the “Ready” message appears under Status; in the Vuser script, an **obj_wait_info** statement waits for the message.

Setting the timeout to thirty seconds ensures that the Vuser waits up to thirty seconds for the message to appear before continuing test execution.

```
# Wait for the message to appear
rc = obj_wait_info("Status","value","Ready.",30);
```

The final section of the test measures the duration of the transaction. An if statement is defined to process the results of the **obj_wait_info** statement.

If the message appears in the field within the timeout, the first **end_transaction** statement records the duration of the transaction and that it passed. If the timeout expires before the message appears, the transaction fails.

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

Index