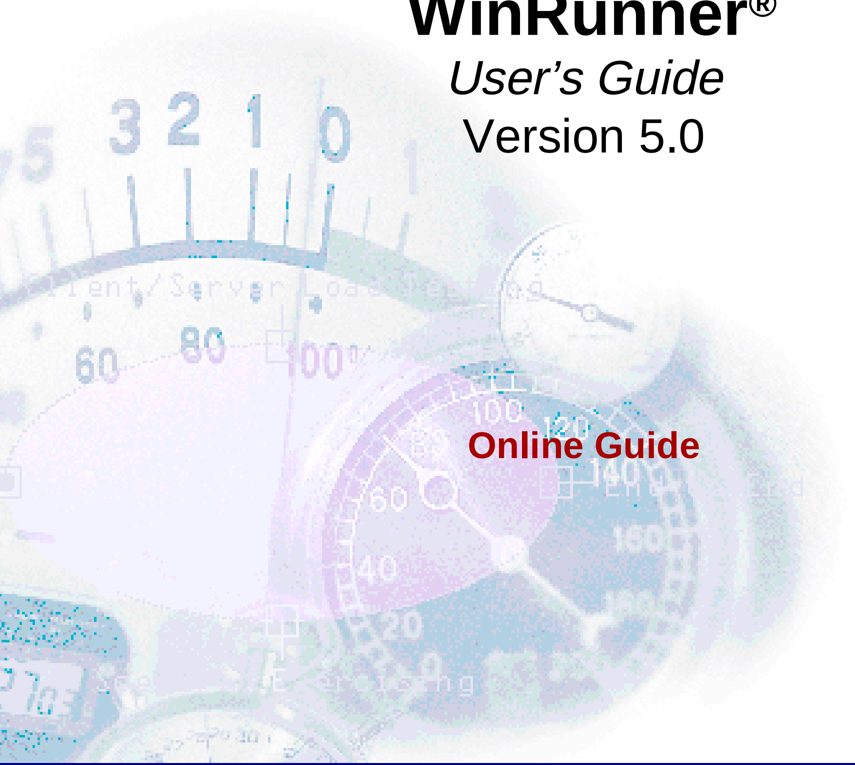
WinRunner®

User's Guide

Version 5.0



Online Guide

Table of Contents

Find

?

Click a page

PART II: UNDERSTANDING THE GUI MAP

Find
?
Click a page

PART III: CREATING TESTS

Find

?

Click a page

Find

?

Click a page

Find
?
Click a page

Find
?
Click a page

Find

?

Click a page

Click a page

Find

?

Click a page

PART VI: DEBUGGING TESTS

Find

?

Click a page

PART VIII: WORKING WITH TESTSUITE

Find

?

Click a page

PART IX: APPENDIX

Find

?

Click a page

Welcome to WinRunner

Welcome to WinRunner, Mercury Interactive's automated GUI testing tool for Microsoft Windows. WinRunner gives you everything you need to quickly create and execute sophisticated automated tests on your application.

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.

This guide contains 8 parts:

Part I: Starting the Testing Process

Provides an overview of WinRunner and the main stages of the testing process.

Part II: Understanding the GUI Map

Describes Context Sensitive testing and the importance of the GUI map for creating adaptable and reusable test scripts.

Part III: Creating Tests

Describes how to create test scripts and insert checkpoints that enable you to check the behavior of your application.

Part IV: Programming with TSL

Describes how to enhance your test scripts using variables, control-flow statements, arrays, user-defined functions, and WinRunner’s visual programming tools.

Part V: Running Tests

Describes how to run your automated tests and analyze test results, and how to report defects detected in your application.

Part VI: Debugging Tests

Describes how you can control test runs in order to identify and isolate bugs in test scripts.

Part VII: Configuring WinRunner

Describes how to change system defaults in order to adapt WinRunner to your testing environment.

Part VIII: Working with TestSuite

Describes how WinRunner interacts with TestDirector and LoadRunner.

WinRunner Documentation Set

In addition to this guide, WinRunner comes with a complete set of documentation:

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

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

Online Resources

WinRunner includes the following online resources:

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

Books Online displays the complete documentation set in PDF format. Online books can be read and printed using Adobe Acrobat Reader 3.01, which is included in the installation package. Check Mercury Interactive's Customer Support web site for updates to WinRunner online books.

WinRunner Context Sensitive Help provides immediate 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. Check Mercury Interactive's Customer Support web site for updates to WinRunner help files.

TSL Online Reference describes Test Script Language (TSL), the functions it contains, and examples of how to use the functions. Check Mercury Interactive's Customer Support site for updates to the *TSL Online Reference*.

WinRunner Sample Tests includes utilities and sample tests with accompanying explanations. Check Mercury Interactive's Customer Support site for updates to WinRunner help files.

Technical Support Online uses your default web browser to open Mercury Interactive's Customer Support web site.

Support Information presents the locations of Mercury Interactive's Customer Support web site and home page, the e-mail address for sending information requests, the name of the relevant news group, the location of Mercury Interactive's public FTP site, and a list of Mercury Interactive's offices around the world.

Mercury Interactive on the Web uses your default web browser to open Mercury Interactive's home page. This site provides you with the most up-to-date information on Mercury Interactive and its products. This includes new software releases, seminars and trade shows, customer support, educational services, and more.

WinRunner Customization Guide explains how to customize WinRunner to meet the special testing requirements of your application. You can download this PDF file from Mercury Interactive's Customer Support web site.

Typographical Conventions

This book uses the following typographical conventions:

Bold	**Bold** text indicates function names and the elements of the functions that are to be typed in literally.
Italics	*Italic* text indicates variable names.
Helvetica	The Helvetica font is used for examples and statements that are to be typed in literally.
[ ]	Square brackets enclose optional parameters.
{ }	Curly brackets indicate that one of the enclosed values must be assigned to the current parameter.
...	In a line of syntax, three dots indicate that more items of the same format may be included. In a program example, three dots are used to indicate lines of a program that were intentionally omitted.
|	A vertical bar indicates that either of the two options separated by the bar should be selected.

Starting the Testing Process

Find

?

Starting the Testing Process

Introduction

Welcome to WinRunner, Mercury Interactive's automated testing tool for Microsoft Windows applications. This guide provides you with detailed descriptions of WinRunner's features and automated testing procedures.

Recent advancements in client/server software tools enable developers to build applications quickly and with increased functionality. Quality Assurance departments must cope with software that has dramatically improved, but is increasingly complex to test. Each code change, enhancement, defect fix, and platform port necessitates retesting the entire application to ensure a quality release. Manual testing can no longer keep pace in this dynamic development environment.

WinRunner helps you to automate the testing process, from test development to execution. You create adaptable and reusable test scripts which challenge the functionality of your application. Prior to a software release, you can execute these tests in a single overnight run—enabling you to detect defects and ensure superior software quality.

WinRunner Testing Modes

WinRunner facilitates easy test creation by recording how you work on your application. As you point and click GUI (Graphical User Interface) objects in your application, WinRunner generates a test script in the C-like Test Script Language (TSL). You can further enhance your test scripts with manual programming. WinRunner includes a Visual Programming tool which helps you to quickly and easily add functions to your recorded tests.

WinRunner includes two modes for recording tests:

Context Sensitive

Context Sensitive mode records your actions on the application being tested in terms of the GUI objects you select (such as windows, lists, and buttons), while ignoring the physical location of an object on the screen. Every time you perform an operation on the application, a TSL statement is generated in the test script that describes the object selected and the action performed.

As you record, WinRunner writes a unique description of each selected object to a GUI map. The GUI map consists of files that are maintained separately from your test scripts. If the user interface of your application changes, you have to update only the GUI map, and not hundreds of tests. This allows you to easily reuse your Context Sensitive test scripts on future versions of your application.

Find
?

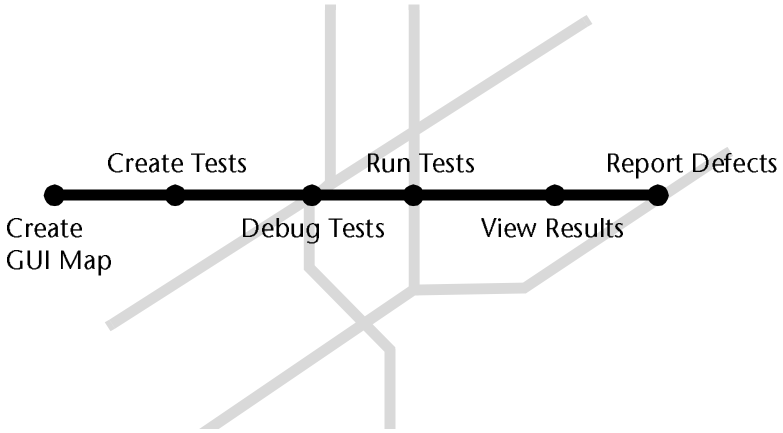
To run a test, you simply play back the test script. WinRunner simulates a human user by moving the mouse pointer over your application, selecting objects, and entering keyboard input. WinRunner reads the object descriptions in the GUI map and then searches in the application under test (AUT) for objects that match these descriptions. It can locate the objects within a window even if their placement has changed.

Analog

Analog mode records mouse clicks, keyboard input, and the exact x- and y-coordinates traveled by the mouse. When the test is executed, WinRunner retraces the mouse tracks. Use Analog mode when exact mouse coordinates are important to your test, such as when testing a drawing application.

The WinRunner Testing Process

Testing with *WinRunner* involves five main stages:



Create the GUI Map

The first stage is to create the GUI map so that WinRunner can recognize the GUI objects in your application. Use the RapidTest Script wizard to review the user interface of your application and systematically add descriptions of every GUI object to the GUI map. Alternatively, you can add descriptions of individual objects to the GUI map by clicking objects while recording a test.

Create Tests

Next, you create test scripts by recording, programming, or a combination of both. While recording tests, insert checkpoints where you want to check the response of the application under test (AUT). You can insert checkpoints that check GUI objects and bitmaps. During this process, WinRunner captures data and saves it as *expected results*—the expected AUT response to the test.

Debug Tests

Run tests in Debug mode to make sure that the test runs smoothly. You can set breakpoints, monitor variables, and control test execution in order to identify and isolate defects. Test results are saved in the debug directory, which you can discard once you have finished debugging the test.

Run Tests

Run tests in Verify mode. Each time WinRunner encounters a checkpoint in the test script, it compares the current data in the AUT to the expected data captured earlier. If any mismatches are found, it captures them as *actual results*.

Analyze Results

Determine the success or failure of the tests. Following each test run, results are displayed in a report. The report details all the major events that occurred during the run, such as checkpoints, error messages, system messages, or user messages.

If mismatches are detected at checkpoints during the test run, then you can view the expected results and the actual results from the Test Results window. In cases of bitmap mismatches, you can also view a bitmap that displays only the difference between the expected and actual results.

Find

?

Report Defects

If a test run fails due to a defect in the application, you can report information about the defect directly from the Test Results window. This information is sent via e-mail to the quality assurance manager, who tracks the defect until it is fixed.

Working with TestSuite

WinRunner works with other TestSuite tools to provide an integrated solution for all phases of the testing process: test planning, test development, GUI and load test execution, defect tracking, and client load testing for multi-user systems.

TestDirector

TestDirector is Mercury Interactive’s software test management tool. It helps quality assurance personnel plan and organize the testing process. With TestDirector you can create a database of manual and automated tests, build test cycles, execute tests, and report and track defects. You can also create reports and graphs to help you review the progress of test planning, execution, and defect tracking before a software release.

When you work with WinRunner, you can choose to save your tests directly to your TestDirector database. You can also execute tests in WinRunner and then use TestDirector to review the overall results of a testing cycle.

LoadRunner

LoadRunner is Mercury Interactive’s testing tool for client/server applications. Using LoadRunner, you can simulate an environment in which many users are simultaneously engaged in a single server application. Instead of human users, it substitutes virtual users that run automated tests on the application being tested. You can test an application’s performance “under load” by simultaneously activating virtual users on multiple host computers.

WinRunner at a Glance

This chapter explains how to start WinRunner and introduces the WinRunner window.

This chapter describes:

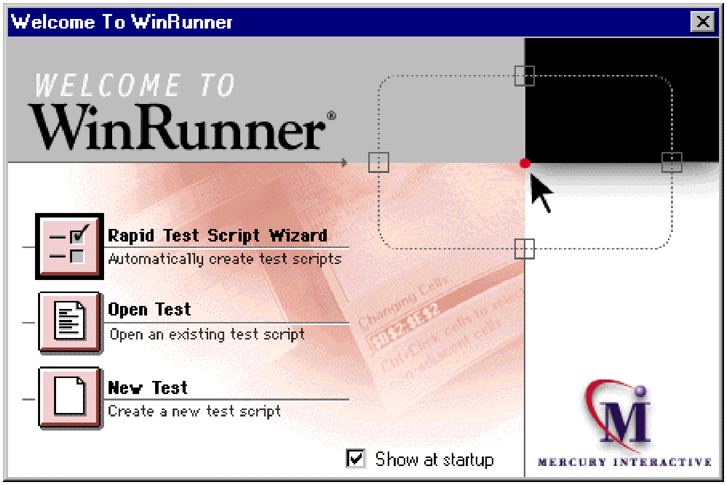
Find

?

Starting WinRunner

To start WinRunner, either click the WinRunner icon in the WinRunner program group or click Programs > WinRunner > WinRunner in the Start menu. After several seconds, the WinRunner window is displayed on your desktop. Notice that the WinRunner Record/Run Engine icon appears in the status area of the Windows taskbar. This engine establishes and maintains the connection between WinRunner and the application you are testing. If you are working with TestDirector, the TdApiWnd icon also appears on your screen.

The first time you start WinRunner, the Welcome to WinRunner window opens. You can choose to run the RapidTest Script wizard, open an existing test, or create a new test.



If you do not want this window to appear the next time you start WinRunner, clear the Show at Startup check box.

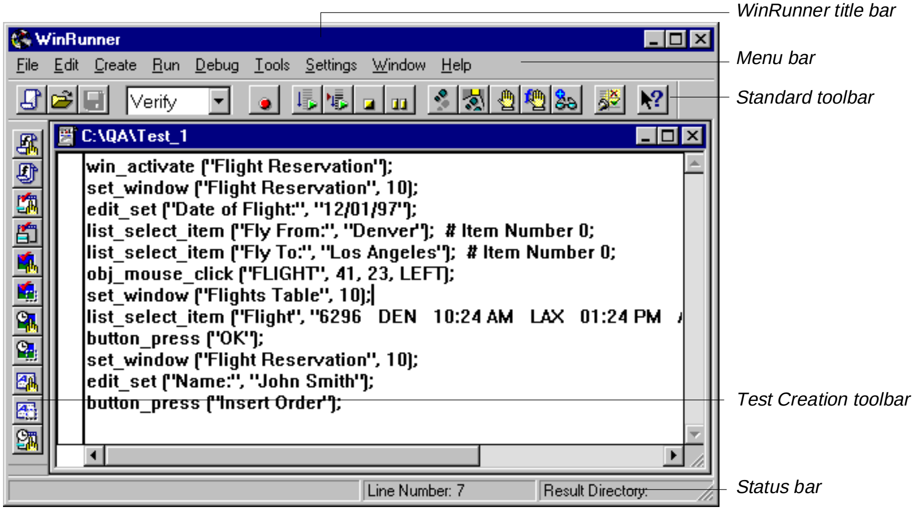
The Main WinRunner Window

The main WinRunner window contains the following key elements:

- *WinRunner title bar*
- *Menu bar,* which displays menus of WinRunner commands
- *Standard toolbar,* which contains the commands you use most frequently when running a test
- *Test Creation toolbar,* which contains the commands you use most frequently while creating a test
- *Status bar*, which displays information on the currently selected command, the line number of the insertion point, and the name of the current results directory

Find

?

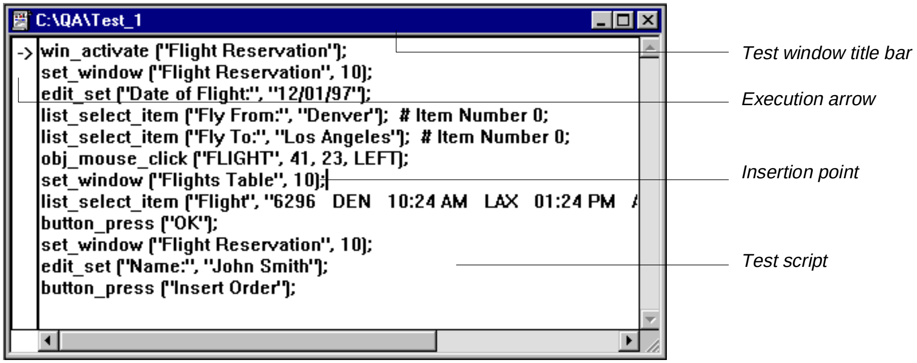


```
win_activate ("Flight Reservation");
set_window ("Flight Reservation", 10);
edit_set ("Date of Flight:", "12/01/97");
list_select_item ("Fly From:", "Denver");  # Item Number 0;
list_select_item ("Fly To:", "Los Angeles");  # Item Number 0;
obj_mouse_click ("FLIGHT", 41, 23, LEFT);
set_window ("Flights Table", 10);
list_select_item ("Flight", "6296   DEN   10:24 AM   LAX   01:24 PM   A
button_press ("OK");
set_window ("Flight Reservation", 10);
edit_set ("Name:", "John Smith");
button_press ("Insert Order");
```

The Test Window

The test window is where you create and run WinRunner tests. It contains the following key elements:

- *Test window title bar*, which displays the name of the open test
- *Test script*, which consists of statements generated by recording and/or programming in TSL, Mercury Interactive's Test Script Language
- *Execution arrow,* which indicates the line of the test script being executed (to move the marker to any line in the script, click the mouse in the left window margin next to the line)
- *Insertion point,* which indicates where you can insert or edit text



```
win_activate ("Flight Reservation");
set_window ("Flight Reservation", 10);
edit_set ("Date of Flight:", "12/01/97");
list_select_item ("Fly From:", "Denver");  # Item Number 0;
list_select_item ("Fly To:", "Los Angeles");  # Item Number 0;
obj_mouse_click ("FLIGHT", 41, 23, LEFT);
set_window ("Flights Table", 10);
list_select_item ("Flight", "6296   DEN   10:24 AM   LAX   01:24 PM   A
button_press ("OK");
set_window ("Flight Reservation", 10);
edit_set ("Name:", "John Smith");
button_press ("Insert Order");
```

Using WinRunner Commands

You can select WinRunner commands from the menu bar or from a toolbar. Certain WinRunner commands can also be executed by pressing softkeys.

Choosing Commands on a Menu

You can choose all WinRunner commands from the menu bar.

Clicking Commands on a Toolbar

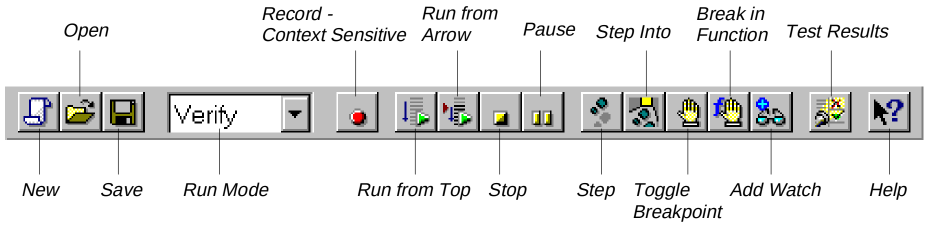
You can execute some WinRunner commands by clicking buttons on the toolbars. WinRunner has two built-in toolbars: the *Standard toolbar* and the *Test Creation toolbar*. You can also create the *User toolbar*, a toolbar you can customize with the commands you most frequently use.

Creating a Floating Toolbar

You can convert each toolbar into a floating toolbar. This enables you to minimize WinRunner to an icon while maintaining access to the commands on a floating toolbar, so that you can work freely with the application you are testing.

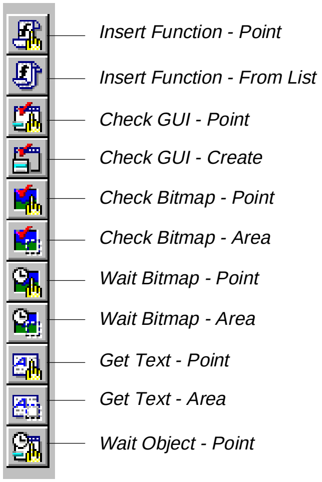
The Standard Toolbar

The Standard toolbar contains buttons for the commands used in running a test. It also contains buttons for opening and saving test scripts, viewing test reports, and accessing help. The default location of the Standard toolbar is below the WinRunner menu bar. For more information about the Standard toolbar, see Chapter 25, **Running Tests**.



The Test Creation Toolbar

The Test Creation toolbar contains buttons for commands used when creating tests. By default, the Test Creation toolbar is hidden. When it is displayed, its default position is at the left edge of the WinRunner window. For more information about the Test Creation toolbar, see Chapter 8, **Creating Tests**.



The User Toolbar

The User toolbar is a customized toolbar that you can create to facilitate access to the commands you most frequently use when testing an application. When it is displayed, the default location of the User toolbar is at the right edge of the WinRunner window. For information on customizing the User toolbar, see **Creating the User Toolbar** on page 550.

Executing Commands Using Softkeys

You can execute some WinRunner commands by pressing softkeys. WinRunner reads input from softkeys even when the WinRunner window is not the active window on your screen, or when it is minimized.

Softkey assignments are configurable. If the application you are testing uses a default softkey that is preconfigured for WinRunner, you can redefine it using WinRunner's softkey configuration utility.

For a list of default WinRunner softkey configurations and information about redefining WinRunner softkeys, see **Configuring WinRunner Softkeys** on page 565.

Find

?

Understanding the GUI Map

Find

?

Understanding the GUI Map
Introducing Context Sensitive Testing

This chapter introduces Context Sensitive testing and explains how WinRunner identifies the Graphical User Interface (GUI) objects in your application.

This chapter describes:

Find

?

About Context Sensitive Testing

Context Sensitive testing enables you to test your application as the user sees it—in terms of GUI objects—such as windows, menus, buttons, and lists. Each object has a defined set of properties that determines its behavior and appearance. WinRunner learns these properties and uses them to identify and locate GUI objects during a test run. WinRunner does not need to know the physical location of a GUI object in order to identify it.

Before you can begin Context Sensitive testing, WinRunner must learn the properties of each GUI object in your application. Use the RapidTest Script wizard to guide you through the learning process. It systematically opens each window in your application and learns the properties of the GUI objects it contains. WinRunner also provides additional methods for learning the properties of individual objects. For more information on the learning process, see Chapter 4, **Creating the GUI Map**.

GUI object properties are saved in the GUI map. WinRunner uses the GUI map to help it locate objects during a test run. It reads an object's description in the GUI map and then looks for an object with the same properties in the application under test (AUT). You can open and view the GUI map in order to gain a comprehensive picture of the objects in your application.

The GUI map ensures that the time and effort you invest in test development is not wasted when you update your application. As the user interface of your application changes, you can continue to use existing tests. You simply add, delete, or edit object descriptions in the GUI map so that WinRunner can continue to find the objects in your application.

How a Test Identifies GUI Objects

You create tests by recording or programming *test scripts*. A test script consists of statements in Mercury Interactive's test script language (TSL). Each TSL statement represents mouse and keyboard input to the application being tested. For more information, see Chapter 8, **Creating Tests**.

WinRunner uses a *logical name* to identify an object: for example "Print" for a Print dialog box, or "OK" for an OK button. This short name connects WinRunner to the object's longer *physical description*. WinRunner uses this detailed description to ensure that each GUI object has its own unique identification. The physical description contains a list of the object's physical properties: the Print dialog box, for example, is identified as a window with the label "Print".

Both the logical name and the physical description form the *GUI map*. The following example illustrates the connection between the logical name and the physical description. Assume that you record a test in which you print a readme file by choosing the Print command on the File menu to open the Print dialog box, and then clicking the OK button. The test script is similar to the following:

```
set_window ("Readme.doc - WordPad", 10);
menu_select_item ("File;Print... Ctrl+P");
set_window ("Print", 12);
button_press ("OK");
```

Find
?

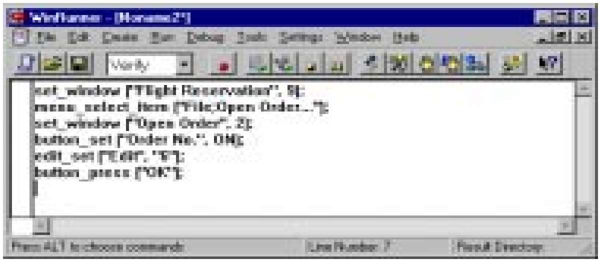
WinRunner learns the actual description—the list of properties and their values—for each of the three objects that are involved and writes this description in the GUI Map:

Readme.doc window: {class: window, label: "Readme.doc - WordPad"}
File menu: {class:menu_item, label:File, parent:None}
Print command: {class: menu_item, label: "Print... Ctrl+P", parent: File}
Print window: {class:window, label:Print}
OK button: {class:push_button, label:OK}

WinRunner also assigns a logical name to each object. As it runs the test, it reads the logical name of each object in the test script and refers to its physical description in the GUI map. WinRunner then uses this description to find the object in the application being tested.

Physical Descriptions

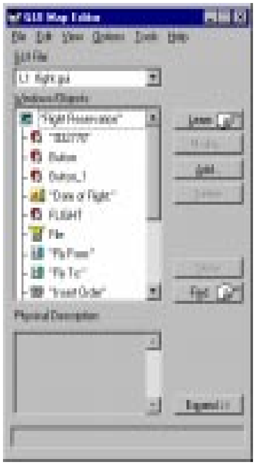
Test Script



```
set_window ("Flight Reservation", 5);
menu_select_item ("File;Open Order...");
set_window ("Open Order", 2);
button_set ("Order No.", ON);
edit_set ("Edit", "5");
button_press ("OK");
```

WinRunner - [Noname2*]
File Edit Create Run Debug Tools Settings Window Help
Verify
Press ALT to choose commands
Line Number: 7
Result Directory

GUI Map

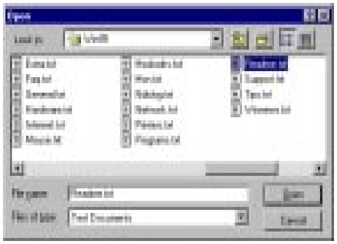


❶ *WinRunner reads the logical name in the test script and refers to the GUI map*

❷ *Matches the logical name with the physical description*

❸ *Uses the physical description to find object in application*

AUT



WinRunner identifies each GUI object in the application under test by its *physical description*: a list of physical properties and their assigned values. These property–value pairs appear in the following format in the GUI map:

{*property1:value1, property2:value2, property3:value3, ...*}

For example, the description of the “Open” window contains two properties: class and label. In this case the class property has the value *window*, while the label property has the value *Open*:

{class:window, label:Open}

The class property indicates the object’s type. Each object belongs to a different class, according to its functionality: window, pushbutton, list, radio button, menu, etc. WinRunner identifies an object by learning its class property.

For each class, there is an accompanying set of default properties, which WinRunner learns. For a detailed description of all properties, see Chapter 6, **Configuring the GUI Map**.

Note that WinRunner always learns the physical description of an object in the context of the window in which it appears. This creates a unique physical description for each object. For more information, see **Setting the Window Context** on page 30.

Logical Names

In the test script, WinRunner does not use the full physical description for an object. Instead, it assigns a short name to each object: the *logical name.*

An object's logical name is determined by its class. In most cases, the logical name is the label that appears on an object: for a button, the logical name is its label, such as OK or Cancel; for a window, it is the window's title bar; and for a list, the logical name is the text that appears next to or above it.

For a static text object, the logical name is a concatenation of the text and the string "(static)". For example, the logical name of the static text "File Name" is: "File Name (static)".

In certain cases, several GUI objects in the same window are assigned the same logical name, using a location selector (for example, LogicalName_1, LogicalName_2). The purpose of the selector property is to create a unique name for the object.

The GUI Map

The GUI map is the sum of one or more GUI map files. These files contain the logical names and physical descriptions of GUI objects. In most cases, you store all the GUI object information for your application in a single GUI map file.

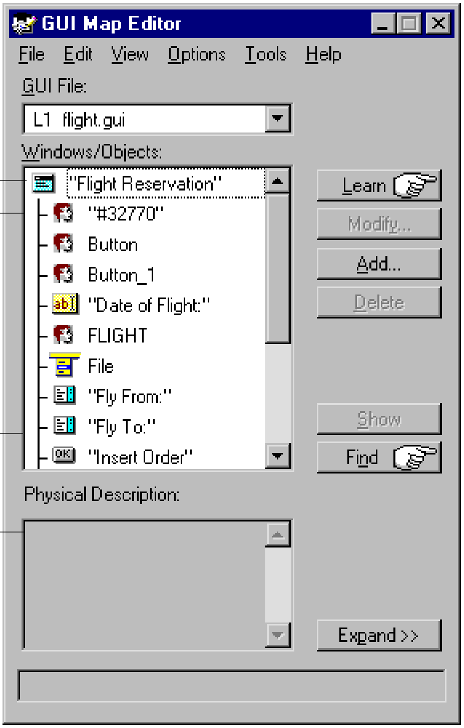
You can view the contents of the GUI map at any time by choosing GUI Map Editor on the Tools menu. In the GUI Map Editor, you can view either the contents of the entire GUI map or the contents of individual GUI map files. GUI objects are grouped according to the window in which they appear in the application.

This view shows the contents of the entire GUI map.

Window

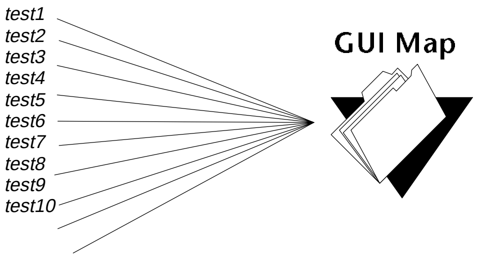
Objects within the window

Expands dialog box to show the physical description of the selected object or window



Find
?

The GUI map enables you to easily keep up with changes made to the user interface of the application being tested. Instead of editing your entire suite of tests, you only have to update the object descriptions in the GUI map.



For example, suppose the OK button in the Open dialog box is changed to a Save button. You do not have to edit every test script that uses this OK button. Instead, you can modify the OK button's physical description in the GUI map, as shown in the example below. The value of the label property for the button is changed from OK to Save:

OK button: {class:push_button, label:Save}

During a test run, when WinRunner encounters the logical name "OK" in the Open dialog box in the test script, it searches for a push button with the label "Save".

You can use the GUI Map Editor to modify the logical names and physical descriptions of GUI objects at any time during the testing process. In addition, you can use the Run wizard to update the GUI map during a test run. The Run wizard opens automatically if WinRunner cannot locate an object in the application being tested. See Chapter 5, **Editing the GUI Map**, for more information.

Setting the Window Context

WinRunner learns and performs operations on objects in the context of the window in which they appear. When you record a test, WinRunner automatically inserts a **set_window** statement into the test script each time the active window changes and an operation is performed on a GUI object. All objects are then identified in the context of that window. For example:

```
set_window ("Print", 12);
button_press ("OK");
```

The **set_window** statement indicates that the Print window is the active window. The OK button is learned within the context of this window.

When programming a test, you need to enter the **set_window** statement manually when the active window changes. When editing a script, take care not to delete necessary **set_window** statements.

Creating the GUI Map

This chapter describes how to teach WinRunner the Graphical User Interface (GUI) of the application being tested and save the information for use during testing.

This chapter describes:

Find

?

About Creating the GUI Map

WinRunner can learn the GUI of your application in several ways. Usually, you use the Script wizard before you start to test in order to learn all the GUI objects in your application at once. This ensures that WinRunner has a complete, well-structured basis for all your Context Sensitive tests. The descriptions of GUI objects are saved in GUI map files. Since all test users can share these files, there is no need for each user to relearn the GUI individually.

If the GUI of your application changes during the software development process, you can use the GUI Map Editor to learn individual windows and objects in order to update the GUI map. You can also learn objects while recording: you simply start to record a test and WinRunner learns the properties of each GUI object you manipulate in your application. This approach is fast and enables a beginning user to create test scripts immediately. However, it is unsystematic, and should not be used as a substitute for the Script wizard if you are planning to develop comprehensive test suites.

You must load the appropriate GUI map files before you run tests. WinRunner uses these files to help locate the objects in the application being tested. The recommended method is to insert a **GUI_load** command into your startup test. When you start WinRunner, it automatically runs the startup test and loads the specified GUI map files. For more information on startup tests, see Chapter 36, **Initializing Special Configurations**. Alternatively, you can insert a **GUI_load** command into individual tests, or use the GUI Map Editor to load GUI map files manually.

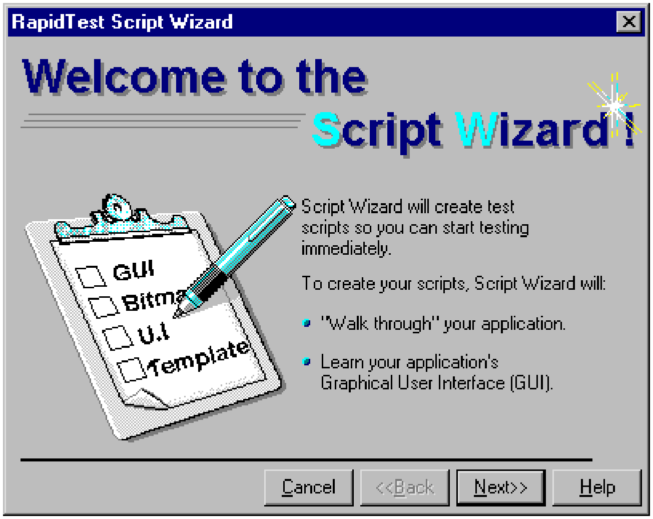
Find
?

Learning the GUI with the RapidTest Script Wizard

The RapidTest Script wizard allows WinRunner to learn all windows and objects in your application at once. It systematically opens every window in the application and learns the GUI objects it contains. WinRunner then instructs you to save the information in a GUI map file. A **GUI_load** command that loads this file is added to a startup test.

To start the RapidTest Script wizard, do one of the following:

- Click Test Wizard in the WinRunner Welcome screen when you start WinRunner.
- Click RapidTest Script Wizard on the Create menu at any time.



Script Wizard will create test scripts so you can start testing immediately.

To create your scripts, Script Wizard will:

- "Walk through" your application.
- Learn your application's Graphical User Interface (GUI).

Learning the GUI by Recording

When you record a test, WinRunner first checks whether the objects you select are in the GUI map. If they are not in the GUI map, WinRunner learns the objects and inserts them into the temporary GUI map file.

In general, you should use recording as a learning tool for small, temporary tests only. Use the Script wizard to learn the entire GUI of your application.

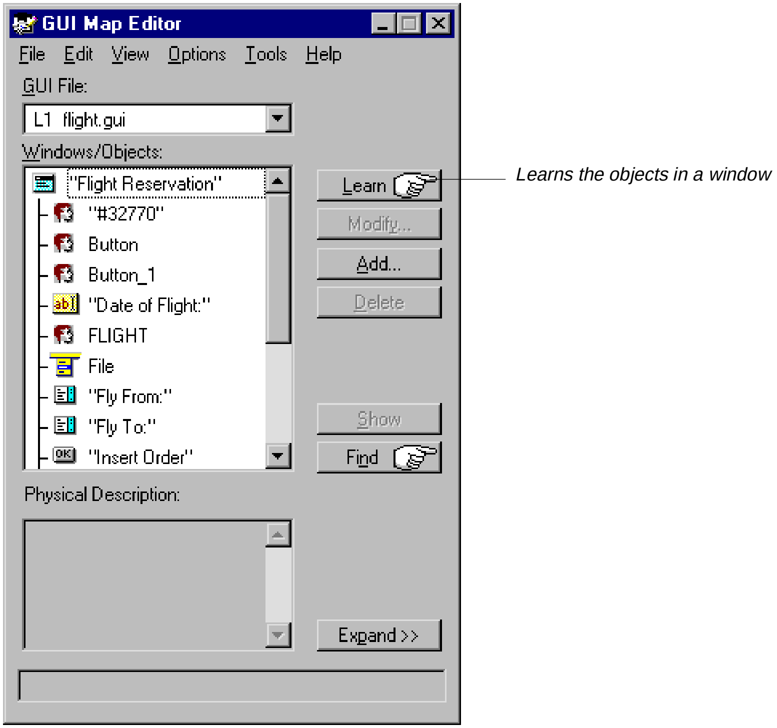
Learning the GUI Using the GUI Map Editor

You can use the GUI Map Editor to learn an individual object or window, or all objects in a window.

To learn GUI objects using the GUI Map Editor:

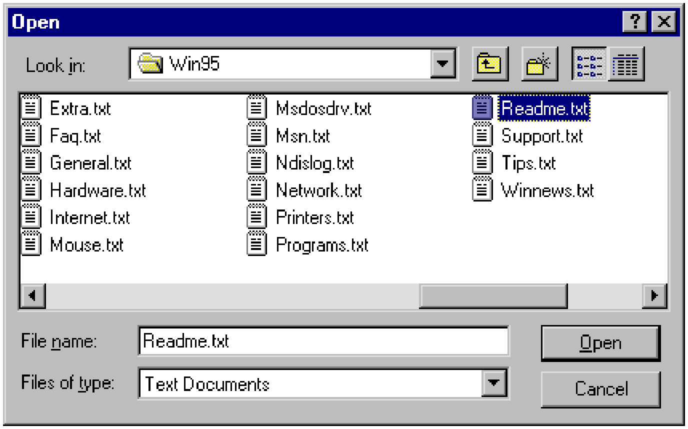
1 On the Tools menu, choose GUI Map Editor. The GUI Map Editor opens.

2 Click Learn. The mouse pointer becomes a pointing hand. (To cancel the operation, click the right mouse button.)


GUI Map Editor
File Edit View Options Tools Help
GUI File:
L1 flight.gui
Windows/Objects:
"Flight Reservation"
"#32770"
Button
Button_1
"Date of Flight:"
FLIGHT
File
"Fly From:"
"Fly To:"
"Insert Order"
Learn
Modify...
Add...
Delete
Show
Find
Physical Description:
Expand >>
Learns the objects in a window

Find
?

3 Place the pointing hand on the object to learn and click the left mouse button. To choose to learn all the objects in a window, place the pointing hand over the window's title bar.



Open

Look in: Win95

Extra.txt
Faq.txt
General.txt
Hardware.txt
Internet.txt
Mouse.txt
Msdosdrv.txt
Msn.txt
Ndislog.txt
Network.txt
Printers.txt
Programs.txt
Readme.txt
Support.txt
Tips.txt
Winnews.txt

File name: Readme.txt

Files of type: Text Documents

GUI information about the learned objects is placed in the active GUI map file. See **Loading the GUI Map File** on page 40 for more information.

Saving the GUI Map

When you learn GUI objects by recording, the object descriptions are added to the temporary GUI map file. The temporary file is always open, so that any objects it contains are recognized by WinRunner. When you start WinRunner, the temporary file is loaded with the contents of the last testing session.

To prevent valuable GUI information from being overwritten during a new recording session, save the temporary GUI map files in a permanent GUI map file.

To save the contents of the temporary file in a permanent GUI map file:

1 On the Tools menu, choose GUI Map Editor. The GUI Map Editor opens.

2 On the View menu, select GUI Files.

3 Make sure that the *<Temporary>* file is displayed in the GUI File list. An asterisk (*) next to the file name indicates that the GUI map file was changed. The asterisk disappears when the file is saved.

4 On the File menu, choose Save to open the Save GUI File dialog box.

5 Click a directory. Type in a new file name or click an existing file.

6 Click OK. The saved GUI map file is loaded and appears in the GUI Map Editor.

You can also move objects from the temporary file to an existing GUI map file. For details, see Chapter 5, **Editing the GUI Map**.

To save the contents of a GUI map file to a TestDirector database:

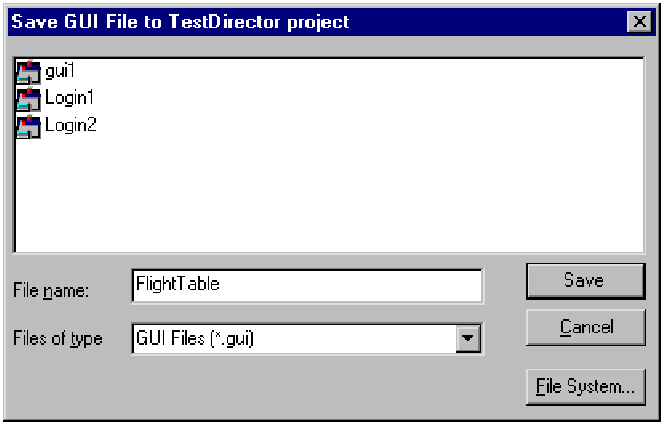
1 Choose Tools > GUI Map Editor to open the GUI Map Editor.

2 On the View menu, select GUI Files.

3 Make sure that the *<Temporary>* file is displayed in the GUI File list. An asterisk (*) next to the file name indicates that the GUI map file was changed. The asterisk disappears when the file is saved.

4 In the GUI Map Editor, choose File > Save.

The Save GUI File to TestDirector Project dialog box opens.


Save GUI File to TestDirector project

gui1
Login1
Login2

File name: FlightTable
Files of type GUI Files (*.gui)

5 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.

6 Click Save to save the GUI map file to a TestDirector database and to close the dialog box.

Note: For more information on saving GUI map files to a TestDirector database, see Chapter 37, **Managing the Testing Process**.

Find

?

Loading the GUI Map File

WinRunner uses GUI map files to locate objects in the application being tested. Before you run tests on your application, you must ensure that the appropriate GUI map files are loaded.

You can load GUI map files in one of two ways:

- using the **GUI_load** command
- from the GUI Map Editor

You can view a loaded GUI map file in the GUI Map Editor. A loaded file is indicated by the letter “L”.

Loading GUI files Using the GUI_load Command

The **GUI_load** statement loads any GUI map file you specify. To load several files, use a separate command for each one. You can insert the **GUI_load** statement at the beginning of any test, but it is preferable to place it in your startup test. This ensures that the GUI map files are loaded automatically each time you start WinRunner. For more information, see Chapter 36, **Initializing Special Configurations**.

To load a file using GUI_load:

1 Open the test from which you want to load the file.

2 Type the **GUI_load** statement as follows, or click the **GUI_load** function in the Function Generator and type in the file path:

GUI_load ("*file_name_full_path*");

For example:

```
GUI_load ("c:\\qa\\flights.gui")
```

See Chapter 17, **Using Visual Programming**, for more information on the Function Generator.

3 Run the test to load the file. See Chapter 25, **Running Tests**, for more information.

Loading GUI Map Files Using the GUI Map Editor

You can load a GUI map file manually, using the GUI Map Editor.

To load a GUI map file using the GUI Map Editor:

1 On the Tools menu, choose GUI Map Editor. The GUI Map Editor opens.

2 On the View menu, select GUI Files.

3 On the File menu, choose Open.

4 In the Open GUI File dialog box, select a GUI map file.

   Note that by default, the file is loaded into the GUI map. If you only want to edit the GUI map file, click Open for Editing Only. See Chapter 5, **Editing the GUI Map**, for more information.

5 Click OK. The GUI map file is added to the GUI file list. The letter "L" indicates that the file has been loaded.

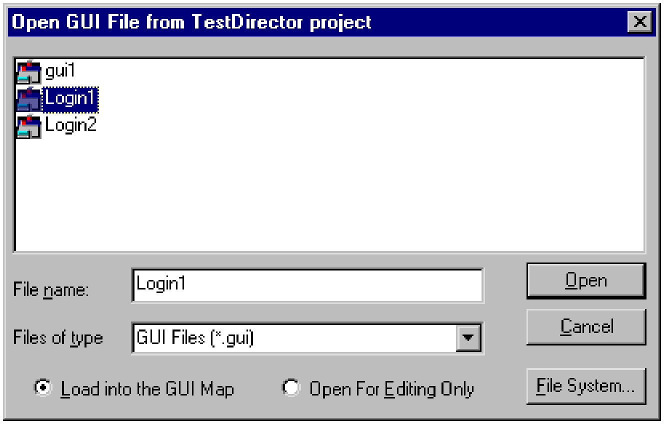
To load a GUI map file from a TestDirector database using the GUI Map Editor:

1 Choose Tools > GUI Map Editor to open the GUI Map Editor.

2 In the GUI Map Editor, choose File > Open.

Find
?

The Open GUI File from TestDirector Project dialog box opens. All the GUI map files that have been saved to the open database are listed in the dialog box.



File name: Login1

Files of type GUI Files (*.gui)

(● Load into the GUI Map) ○ Open For Editing Only

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 text 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, see Chapter 5, **Editing the GUI Map**.

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.

Find

?

Note: For more information on loading GUI map files from a TestDirector database, see Chapter 37, **Managing the Testing Process**.

Editing the GUI Map

This chapter describes how you can extend the life of your tests by modifying descriptions of objects in the GUI map.

This chapter describes:

About Editing the GUI Map

WinRunner uses the GUI map to identify and locate GUI objects in your application. If the GUI of your application changes, you must update object descriptions in the GUI map so that you can continue to use existing tests.

You can update the GUI map in two ways:

- during a test run, using the Run wizard
- at any time during the testing process, using the GUI Map Editor

The Run wizard opens automatically during a test run if WinRunner cannot locate an object in the application being tested. It guides you through the process of identifying the object and updating its description in the GUI map. This ensures that WinRunner will find the object in subsequent test runs.

You can also manually edit the GUI map using the GUI Map Editor. You can modify the logical names and physical descriptions of objects, add new descriptions, and remove obsolete descriptions. You can also move or copy descriptions from one GUI map file to another.

Note that before you can update the GUI map, the appropriate GUI map files must be loaded. You can load files by using the **GUI_load** statement in a test script or by choosing File > Open in the GUI Map Editor. See Chapter 4, **Creating the GUI Map**, for more information.

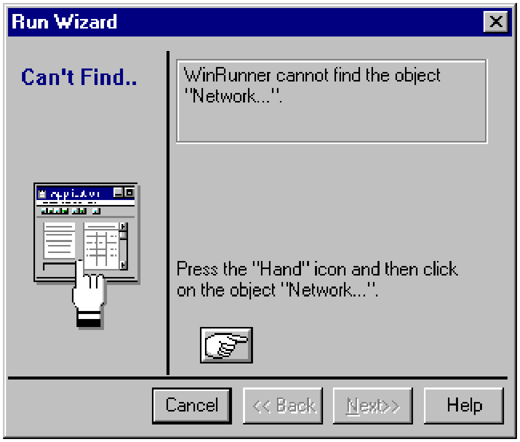
The Run Wizard

The Run wizard detects changes in the GUI of your application that interfere with test execution. During a test run, it automatically opens when WinRunner cannot locate an object. The Run wizard prompts you to point to the object in your application, determines why the object cannot be found, and then offers a solution. The Run wizard suggests loading an appropriate GUI map file; in most cases, a new description is automatically added to the GUI map or the existing description is modified. When this process is completed, the test run continues. In future test runs, WinRunner can successfully locate the object.

For example, suppose you run a test in which you click the Network button in an Open window.

```
set_window ("Open");
button_press ("Network");
```

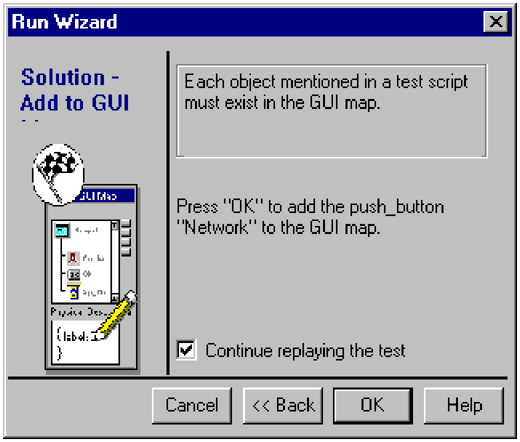
If the Network button is not in the GUI map, then the Run wizard opens and describes the problem.



WinRunner cannot find the object "Network...".

Press the "Hand" icon and then click on the object "Network...".

Click the Hand button in the wizard and point to the Network button. The Run wizard offers a solution.


Run Wizard

Solution - Add to GUI

Each object mentioned in a test script must exist in the GUI map.

Press "OK" to add the push_button "Network" to the GUI map.

[✓] Continue replaying the test

When you click OK, the Network object description is automatically added to the GUI map and WinRunner resumes test execution. The next time you run the test, WinRunner can identify the Network button.

Note that in some cases, the Run wizard edits the test script, not the GUI map. For example, if WinRunner cannot locate an object because the appropriate window is inactive, then the Run wizard inserts a **set_window** statement in the test script.

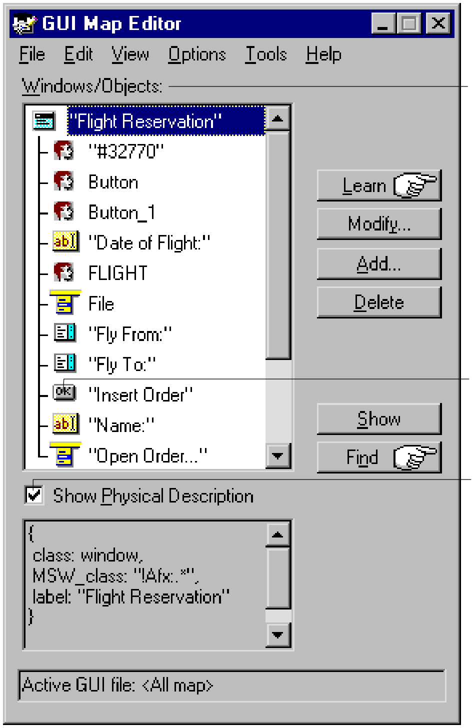
The GUI Map Editor

You can edit the GUI map at any time using the GUI Map Editor. To open the GUI Map Editor, choose GUI MapEditor on the Tools menu.

Two views are available in the GUI Map Editor. By default, the contents of the entire GUI map are displayed.

Find

?



```
{
 class: window,
 MSW_class: "!Afx:.*",
 label: "Flight Reservation"
}
```

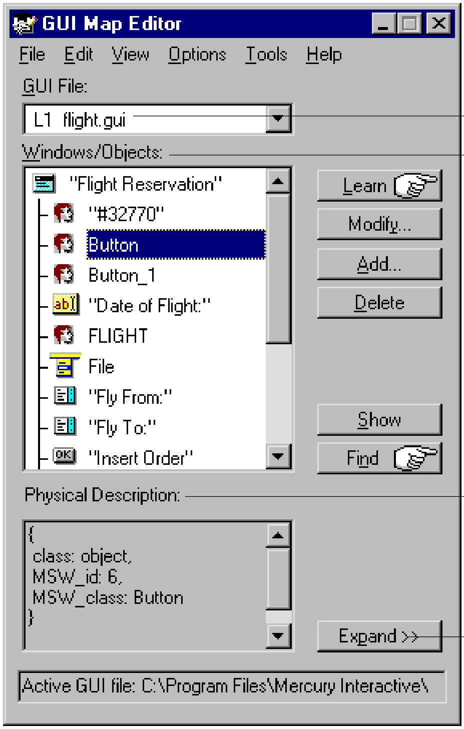
Active GUI file: <All map>

Shows all the windows and objects in the GUI map.

Objects within windows are indented.

When selected, displays the physical description of the selected object or window.

When viewing the contents of specific GUI map files, you can expand the GUI Map Editor to view two GUI map files simultaneously. This allows you to easily copy or move descriptions between files. To view the contents of individual GUI map files, select GUI Files from the View menu.



```
{
 class: object,
 MSW_id: 6,
 MSW_class: Button
}
```

Lists the open GUI map files.

Shows the windows and objects in the currently displayed GUI map file.

Displays the physical description of the selected window or object.

Expands the dialog box so you can view the contents of two GUI map files.

In the GUI Map Editor, objects are displayed in a tree under the window in which they appear. When you double-click in a window in the tree, you can view all the objects it contains. To concurrently view all the objects in the tree, choose Expand Tree on the View menu. To view windows only, choose Collapse Tree.

When you view the entire GUI map, you can select the Show Physical Description check box to display the physical description of any object you select in the Windows/Objects list. When you view the contents of a single GUI map file, the GUI Map Editor automatically displays the physical description.

For example, if you select Show Physical Description and click the WordPad window in the window list, the following physical description is displayed in the lower part of the GUI Map Editor:

```
{
class: window,
label: "Document - WordPad",
MSWclass: WordPadClass
}
```

Note that if the value of a property contains any spaces or special characters (such as exclamation marks and asterisks, as in the above example), that value must be enclosed by quotation marks.

Modifying Logical Names and Physical Descriptions

You can modify the logical name or the physical description of an object in a GUI map file using the GUI Map Editor.

Changing the logical name of an object is useful when the logical name that is assigned is not sufficiently descriptive or is too long. For example, suppose WinRunner assigns the logical name Employee Address (static) to a static text object. You can change the name to Address to make the scripts easier to read.

Changing the physical description is necessary when a property value of an object changes. For example, suppose that the label of a button is changed from Insert to Add. You can modify the value of the label property in the physical description of the Insert button as shown below:

```
Insert button:{class:push_button, label:Add}
```

During a test run, when WinRunner encounters the logical name Insert in a test script, it searches for the button with the label Add.

To modify an object's logical name or physical description in a GUI map file:

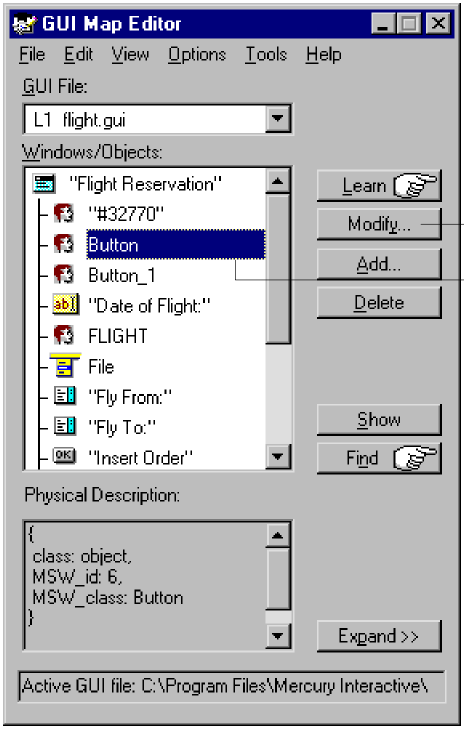
1 Choose GUI Map Editor on the Tools menu to open the GUI Map Editor.

2 Select GUI Files from the View menu.

3 If the appropriate GUI map file is not loaded, select File > Open to open the file.

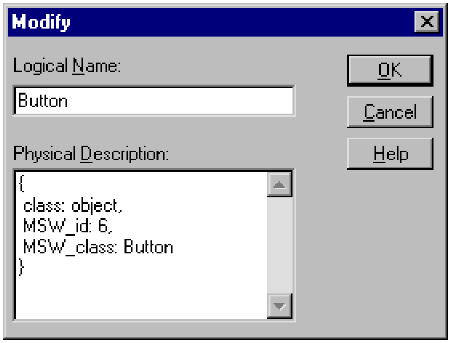
4 To see the objects in a window, double-click the window name in the Windows/Objects field. Note that objects within a window are indented.

5 Select the window or object to modify.



```
{
 class: object,
 MSW_id: 6,
 MSW_class: Button
}
```

6 Click Modify to open the Modify dialog box.



Logical Name: Button

Physical Description:

```
{
 class: object,
 MSW_id: 6,
 MSW_class: Button
}
```

7 Edit the logical name or physical description as desired and click OK. The change appears immediately in the GUI map file.

How WinRunner Handles Varying Window Labels

Windows often have varying labels. For example, the main window in a text application might display a file name in the title bar as well as the application name. In order to avoid a situation in which a window is not recognized because its name changed after it was learned, WinRunner defines a regular expression in the physical description of windows with varying labels.

WinRunner inserts a regular expression in the description of all windows that it learns, with the exception of dialog boxes. When the label of a window contains a dash, the wildcard character (*) is placed following the dash; otherwise, it is placed at the end of the label property value.

For example, suppose WinRunner learns a description of the main window in the Windows 95 WordPad application. The following physical description is learned:

```
{
class: window,
label: "Document - WordPad"
MSWclass: WordPadClass
}
```

The regular expression enables WinRunner to recognize the Notepad window regardless of the name that appears after the dash in the window title.

Changing Regular Expressions in the Physical Description

WinRunner uses two “hidden” properties in order to use a regular expression in the physical description of an object. These properties are regexp_label and regexp_MSW_class.

The regexp_label property is used for windows only. It operates “behind the scenes” to insert a regular expression into a window’s label description. Note that when using WinRunner for Windows 95, this property is not obligatory, and therefore it is neither recorded nor learned.

The regexp_MSW_class property inserts a regular expression into an object’s MSW_class. It is obligatory for all types of windows and for the object class.

Adding a Regular Expression

You can add the regexp_label and the regexp_MSW_class property as needed to the GUI configuration for a class. You would add a regular expression in this way when either the label or the MSW class of objects in your application has characters in common that can safely be ignored.

Suppressing a Regular Expression

You can suppress the use of a regular expression in the physical description of a window. Suppose the label of all the windows in your application begins with “AAA Wingnuts — ”. To ensure that WinRunner can distinguish between the windows, you could replace the regexp_label property in the list of obligatory learned properties for windows in your application with the label property. See Chapter 6, **Configuring the GUI Map**, for more information.

Find

?

Copying and Moving Objects between Files

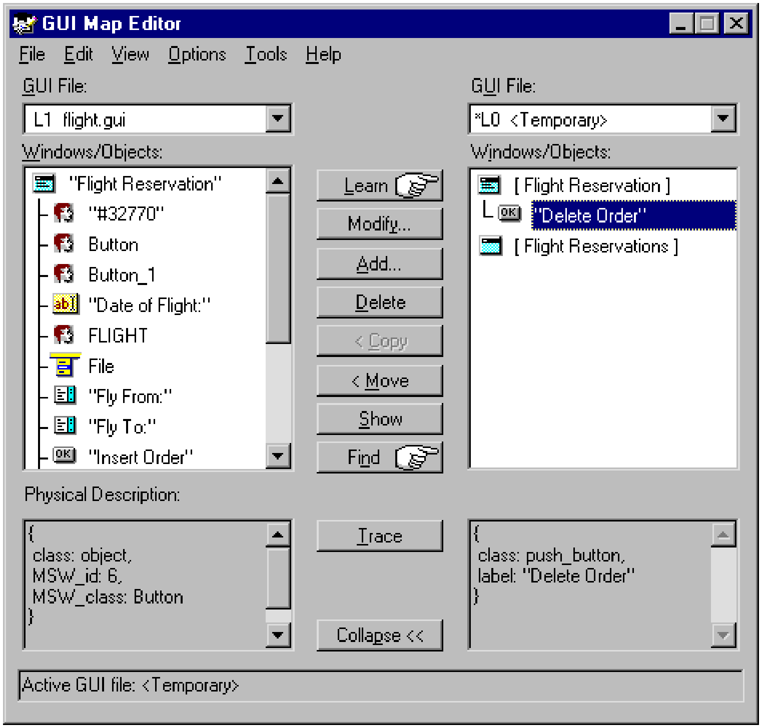
You can update GUI map files by copying or moving the description of GUI objects from one GUI map file to another. Note that you can only copy objects from a GUI file that you have opened for editing only.

To copy or move objects between two GUI map files:

1 On the Tools menu, choose GUI Map Editor to open the GUI Map Editor.

2 From the View menu, select GUI Files.

3 Click Expand in the GUI Map Editor. The dialog box expands to display two GUI map files simultaneously.



```
{
 class: object,
 MSW_id: 6,
 MSW_class: Button
}
```

```
{
 class: push_button,
 label: "Delete Order"
}
```

Active GUI file: <Temporary>

4 View a different GUI map file on each side of the dialog box by clicking the file names in the GUI File lists.

5 In one file, select the objects you want to copy or move. Use the Shift key to select multiple objects. To select all objects in a window, choose Select All on the Edit menu.

6 Click Copy or Move.

7 To restore the GUI Map Editor to its original size, click Collapse.

Find

?

Finding an Object in a GUI Map File

You can easily find the description of a specific object in a GUI map file by pointing to the object in the application being tested.

To find an object in a GUI map file:

1 Choose GUI Map Editor on the Tools menu to open the GUI Map Editor.

2 Select GUI Files from the View menu.

3 Select File > Open to load the GUI map file.

4 Click Find. The mouse pointer turns into a pointing hand.

5 Click the object in the application being tested. The object is highlighted in the GUI map file.

Finding an Object in Multiple GUI Map Files

If an object is described in more than one GUI map file, you can quickly locate all the object descriptions using the Trace button in the GUI Map Editor. This is particularly useful if you learn a new description of an object and want to find and delete older descriptions in other GUI map files.

To find an object in multiple GUI map files:

1 Choose GUI Map Editor on the Tools menu to open the GUI Map Editor.

2 Select GUI Files from the View menu.

3 Open the GUI Map files in which the object description might appear.

For each file, select File > Open to open the Open - GUI File dialog box. Choose the GUI map file you want to open and click Open for Editing Only. Click OK.

4 Display the contents of the file with the most recent description of the object.

5 Select the object in the Windows/Objects field.

6 Click the Expand button to expand the GUI Map Editor dialog box.

7 Click the Trace button. The GUI map file in which the object is found is displayed on the other side of the dialog box, and the object is highlighted.

Manually Adding an Object to a GUI Map File

You can manually add an object to a GUI map file by copying the description of another object, and then editing it as needed.

To manually add an object to a GUI map file:

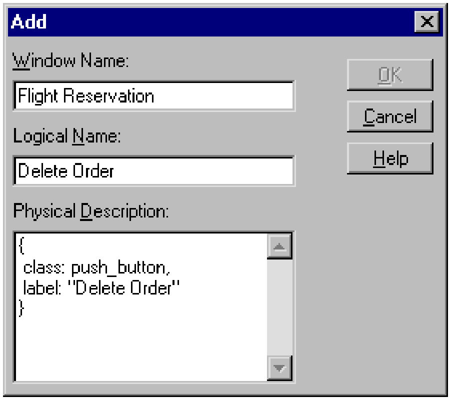
1 Choose GUI Map Editor on the Tools menu to open the GUI Map Editor.

2 Select GUI Files from the View menu.

3 Select File > Open to open the appropriate GUI map file.

4 Select the object to use as the basis for editing.

5 Click the Add button to open the Add dialog box.



Window Name:
Flight Reservation

Logical Name:
Delete Order

Physical Description:
```
{
 class: push_button,
 label: "Delete Order"
}
```

6 Edit the appropriate fields and click OK. The object is added to the GUI map file.

Deleting an Object from a GUI Map File

If an object description is no longer needed, you can delete it from the GUI map file.

To delete an object from a GUI map file:

1. Choose GUI Map Editor on the Tools menu to open the GUI Map Editor.
2. Select GUI Files from the View menu.
3. Select File > Open in the GUI Map Editor to open the appropriate GUI map file.
4. Select the object to be deleted. If you want to delete more than one object, use the Shift key to make your selection.
5. Click the Delete button.
6. Select File > Save to save the changes to the GUI map file.

To delete all objects in a window:

1. Choose GUI Map Editor on the Tools menu to open the GUI Map Editor.
2. Select GUI Files from the View menu.
3. Select File > Open in the GUI Map Editor to open the appropriate GUI map file.
4. Choose Clear All from the Edit menu.

Clearing a GUI Map File

You can quickly clear the entire contents of the temporary GUI map file, or any other GUI map file.

To delete the entire contents of a GUI map file:

1. Choose GUI Map Editor on the Tools menu to open the GUI Map Editor.
2. Select GUI Files from the View menu.
3. Open the appropriate GUI map file.
4. Display the GUI map file at the top of the GUI File list.
5. Choose Clear All from the Edit menu.

Filtering Displayed Objects

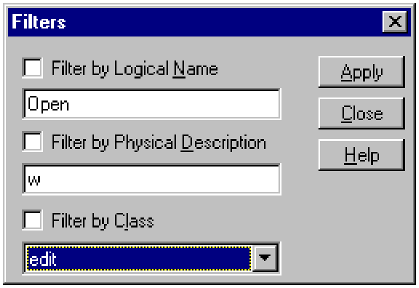
You can filter the list of objects displayed in the GUI Map Editor by using any of the following filters:

- *Logical name* displays only objects with the specified logical name or substring (for example, “Open” or “Op”).
- *Physical description* displays only objects that match the specified physical description. Use any substring belonging to the physical description (for example, specifying “w” filters out all objects that contain a “w” in their physical description).
- *Class* displays only the objects of the specified class, such as all the pushbuttons.

To apply a filter:

1 Choose GUI Map Editor on the Tools menu to open the GUI Map Editor.

2 Choose Filters on the Options menu to open the Filters dialog box.


Filters

☐ Filter by Logical Name: Open

☐ Filter by Physical Description: w

☐ Filter by Class: edit

3. Select the type of filter you want by selecting a check box and entering the appropriate information.
4. Click the Apply button. The GUI Map Editor displays objects according to the filter applied.

Saving Changes to the GUI Map

If you edit the logical names and physical descriptions of objects in the GUI map, you must save the changes in the GUI Map Editor before ending the testing session and exiting WinRunner.

To save changes to the GUI map, do one of the following:

- Select File > Save in the GUI Map Editor to save changes in the appropriate GUI map file.
- Select File > Save As to save the changes in a new GUI map file.

Configuring the GUI Map

This chapter describes how you can change the way WinRunner identifies GUI objects during Context Sensitive testing.

This chapter describes:

About Configuring the GUI Map

Each GUI object in the application under test is defined by multiple properties, such as class, label, MSW_class, MSW_id, x, y, 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 of its properties. Instead, it learns the minimum number of properties that enables 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.

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

Find
?

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 will activate this configuration.

Note: If your application contains owner-drawn custom buttons, you can map all of them to one of the standard button classes instead of mapping each button separately. You do this in one of two ways: by choosing a standard button class in the Record Owner-Drawn Buttons box in the Record tab in the Options dialog box; or by setting the rec_owner_drawn testing option with the **setvar** function from within a test script. For more information, see Chapter 34, **Setting Testing Options from a Test Script**.

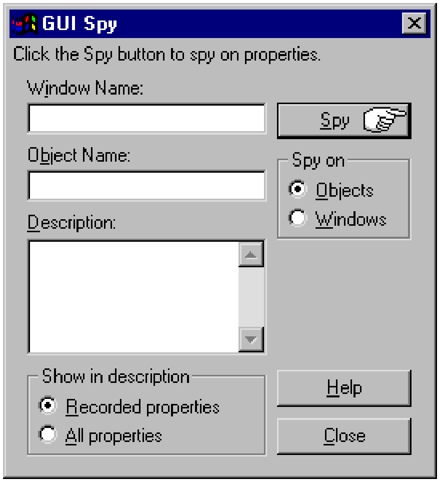
Object properties vary in their degree of portability. Some are non-portable (unique to a specific platform), such as MSW_class or MSW_id. Some are semi-portable (supported by multiple platforms, but with a value that is likely to change), such as handle, or Toolkit_class. Others are fully portable (such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

Viewing GUI Object Properties

Using the GUI Spy, you can view the properties of any GUI object on your desktop. You simply point to an object and the GUI Spy displays the properties and their values in the GUI Spy dialog box. You can choose to view all the properties of an object, or only the default set of properties defined for the object class.

To spy on a GUI object:

1 On the Tools menu, choose GUI Spy to open the GUI Spy dialog box.



Click the Spy button to spy on properties.

Show in description: [✓ Recorded properties] All properties

By default, the GUI Spy displays the properties of objects within windows. To view the properties of a window, click Windows in the Spy On box.

2 To view all the properties defined for an object, Click All Properties in the Show in Description box. If the All Properties option is not selected, the GUI Spy displays only the default set of properties for an object.

3 Click Spy and point to an object on the screen. The object is highlighted and the active window name, object name, and object description (properties and their values) appear in the appropriate fields.

Note that as you move the pointer over other objects, each one is highlighted in turn and its description appears in the dialog box.

4 To capture an object description in the GUI Spy dialog box, point to the desired object and press the STOP softkey.

5 Click Close to exit the dialog box.

Understanding the Default GUI Map Configuration

For each class, WinRunner learns a set of default properties. Each default property is classified as either obligatory or optional. (For a list of the default properties, see **All Properties** on page 94.)

- An *obligatory* property is one that is always learned (if it exists).
- An *optional* property is used only if the obligatory properties do not provide a unique identification of an object. 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 a 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 assigned to an object by the application developer.

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 by the developer to identify the objects in a window. This selector is recommended if the location of objects with the same description may change within a window. See **Configuring a Standard or Custom Class** on page 81 for more information.

Mapping a Custom Object to a Standard Class

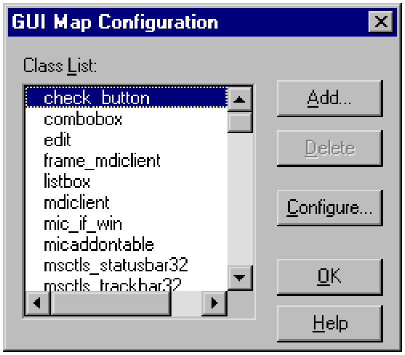
A custom object is any GUI object that does not belong to one of the standard classes used by WinRunner. WinRunner learns such objects under the generic “object” class. WinRunner records operations on custom objects using **obj_mouse_** statements.

Using the GUI Map Configuration dialog boxes, 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 Borbtn 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.

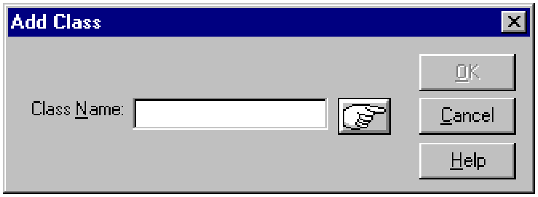
To map a custom object to a standard class:

1 On the Tools menu, choose GUI Map Configuration to open the GUI Map Configuration dialog box.



The Class List displays all the standard and custom classes identified by WinRunner.

2 Click Add to open the Add Class dialog box.

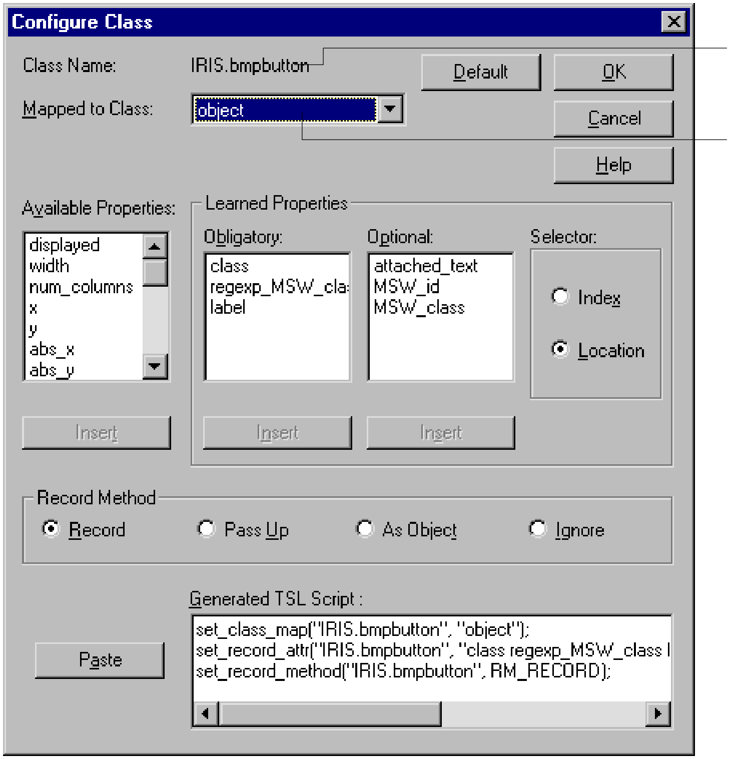


Find
?

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 Configure to open the Configure Class dialog box.


Configure Class

Class Name: IRIS.bmpbutton

Mapped to Class: object

Available Properties: displayed, width, num_columns, x, y, abs_x, abs_y

Learned Properties

Obligatory: class, regexp_MSW_cla, label

Optional: attached_text, MSW_id, MSW_class

Selector: ( ) Index [✓ Location]

Record Method: [✓ Record] ( ) Pass Up ( ) As Object ( ) Ignore

Generated TSL Script :

```
set_class_map("IRIS.bmpbutton", "object");
set_record_attr("IRIS.bmpbutton", "class regexp_MSW_class l
set_record_method("IRIS.bmpbutton", RM_RECORD);
```

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, click 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 81.

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 88 for more information.

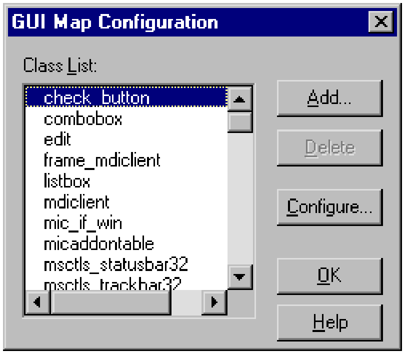
Configuring a Standard or Custom Class

For any of the standard or custom classes, you can modify the following:

- the properties learned
- the selector
- the record method

To configure a standard or custom class:

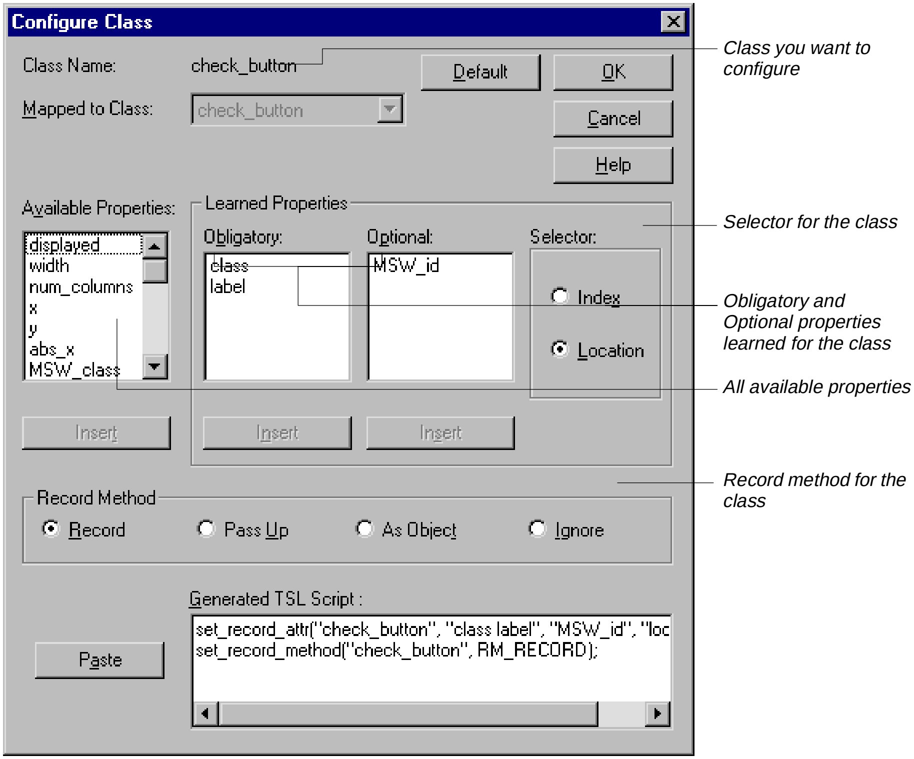
1 On the Tools menu, choose GUI Map Configuration to open the GUI Map Configuration dialog box.



check_button
combobox
edit
frame_mdiclient
listbox
mdiclient
mic_if_win
micaddontable
msctls_statusbar32
msctls_trackbar32

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

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



Selector: Index [✓ Location]

Record Method: [✓ Record] Pass Up As Object Ignore

```
set_record_attr("check_button", "class label", "MSW_id", "loc
set_record_method("check_button", RM_RECORD);
```

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 record method as desired. See **Configuring Learned Properties** on page 84, **Configuring the Selector** on page 87, and **Configuring the Recording Method** on page 88 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 88 for more information.

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 moving properties from one list to another within the dialog box in order to specify whether they are obligatory, optional, or available. Each property can appear in only one of the lists.

- The Obligatory list contains properties that are always learned (provided that they are valid for the specific object).
- The Optional list contains properties that 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 object in the list.
- The Available Properties list contains all remaining properties that are not in either of the other two lists.

When the dialog box is displayed, the Obligatory and Optional lists display the properties learned for the class that appears in the Class Name field.

To modify the property configuration:

1 Click a property to move from any of the lists. Then click the Insert button under the target list. For example:

To move the *MSW_class* property from the Obligatory list to the Optional list, click it in the Obligatory list, then click Insert under the Optional list.

To remove a property so that it is not learned, click it in the Obligatory or Optional list, then click Insert under the Available Properties list.

2 To modify the order of properties within a list (particularly important in the Optional list), click one or more properties and click Insert under the same list. The properties are moved to the bottom of 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, (object)
class	All classes
displayed	All classes
enabled	All classes
focused	All classes
handle	All classes

Property	Classes
height	All classes
label	push_button, radio_button, check_button, window, static, object
maximizable	window
minimizable	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
value	check_button, radio_button, combobox, listbox, scrollbar, static, (object)

Property	Classes
vb_name	All classes
virtual	list, push_button, radio_button, table, object (virtual objects only)
width	All classes
x	All classes
y	All classes

Find

?

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 74.

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 record 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

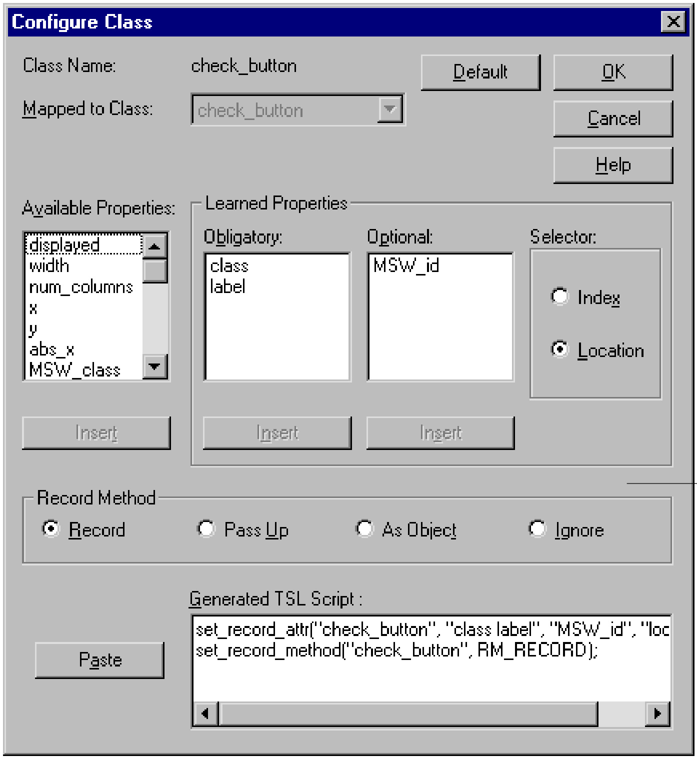
By generating TSL statements that describe the configuration you set and inserting them into a startup test, you can ensure that WinRunner always uses the correct GUI map configuration for your standard and custom object classes.

To create a permanent GUI map configuration for a class:

1 On the Tools menu, choose GUI Map Configuration to open the GUI Map Configuration dialog box.

2 Click a class and click Configure. The Configure Class dialog box appears.

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



Selector: Index [✓ Location]

Record Method: [✓ Record] Pass Up As Object Ignore

```
set_record_attr("check_button", "class label", "MSW_id", "loc
set_record_method("check_button", RM_RECORD);
```

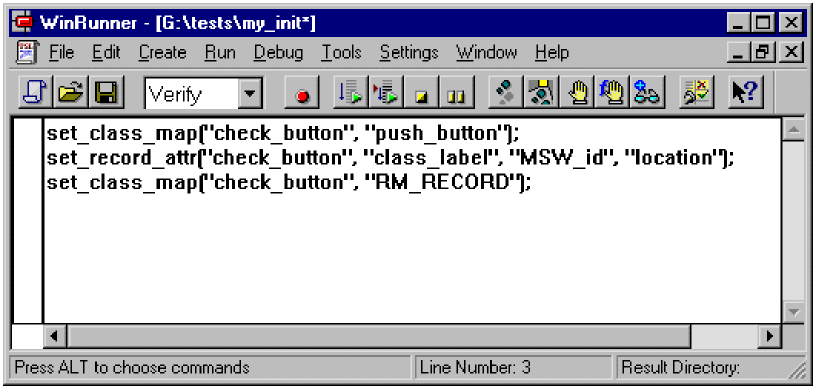
TSL statements describing the GUI map configuration

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 directory), your startup test is defined as follows:

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

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


WinRunner - [G:\tests\my_init*]
File Edit Create Run Debug Tools Settings Window Help
Verify

```
set_class_map("check_button", "push_button");
set_record_attr("check_button", "class_label", "MSW_id", "location");
set_class_map("check_button", "RM_RECORD");
```

Press ALT to choose commands
Line Number: 3
Result Directory:

For more information on startup tests, see Chapter 36, **Initializing Special Configurations**. For more information on the TSL functions that define a custom GUI map configuration (**set_class_map**, **set_record_attr**, and **set_record_method**), refer to the *TSL Online 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 On the Tools menu, choose GUI Map Configuration to open the GUI Map Configuration dialog box.

2 Click the class you want to delete from the Class list.

3 Click Delete.

The Class Property

The class property is the primary property that WinRunner uses to identify the class of a GUI object. WinRunner categorizes GUI objects according to the following classes:

Class	Description
check_button	A checkbox
edit	An edit field
frame_mdiclient	Enables WinRunner to treat a window as an mdiclient object.

Class	Description
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	All objects 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 that is 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

All Properties

The following tables list all properties used by WinRunner in Context Sensitive testing. Properties are listed according to their portability levels: portable, semi-portable, and non-portable.

Note: You cannot use GUI maps that were created in XRunner in WinRunner test scripts. You must create new GUI maps in WinRunner. For information on running XRunner test scripts recorded in Analog mode, see Chapter 8, **Creating Tests**. For information on using GUI checkpoints created in XRunner in WinRunner test scripts, see Chapter 9, **Checking GUI Objects**. For information on using bitmap checkpoints created in XRunner in WinRunner test scripts, see Chapter 12, **Checking Bitmaps**.

Portable Properties

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
attached_text	The static text located near the object
class	See **The Class Property** on page 92.
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
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

Property	Description
minimizable	A Boolean value indicating whether a window can be minimized: 1 if the window can be minimized, 0 if not
nchildren	The number of children the object has: the total number of descendants of the object
parent	The logical name of the parent of the object
position	The position (top to bottom) of a menu item within the menu (the first item is at position 0)
submenu	A Boolean value indicating whether a menu item has a submenu: 1 if menu has submenu, 0 if not
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.
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

Find

?

Semi-Portable Properties

Property	Description
handle	A run-time pointer to the object: the HWND handle
TOOLKIT_clas s	The value of the 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.

Non-Portable Microsoft Windows Properties

Property	Description
active	A Boolean value indicating whether this is the top-level window associated with the input focus
MSW_class	The Microsoft Windows class
MSW_id	The Microsoft Windows ID
obj_col_nam e	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

Find

?

Property	**Description**
pb_name	A text string assigned to PowerBuilder objects by the developer. (The property applies only to WinRunner with PowerBuilder add-in support.)
regexp_label	The text string and regular expression that enables WinRunner to identify an object with a varying label
regexp_MSW class	The Microsoft Windows class combined with a regular expression. Enables WinRunner to identify objects with a varying MSW_class.
sysmenu	A Boolean value indicating whether a menu item is part of a system menu
text	The visible text in an object or window
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)

Find

?

Default Properties Learned

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.

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

Properties for PowerBuilder Objects

The following table lists the standard object classes and the properties learned for each PowerBuilder object.

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 your *WinRunner Installation Guide*.

Understanding the GUI Map
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 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 95/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.

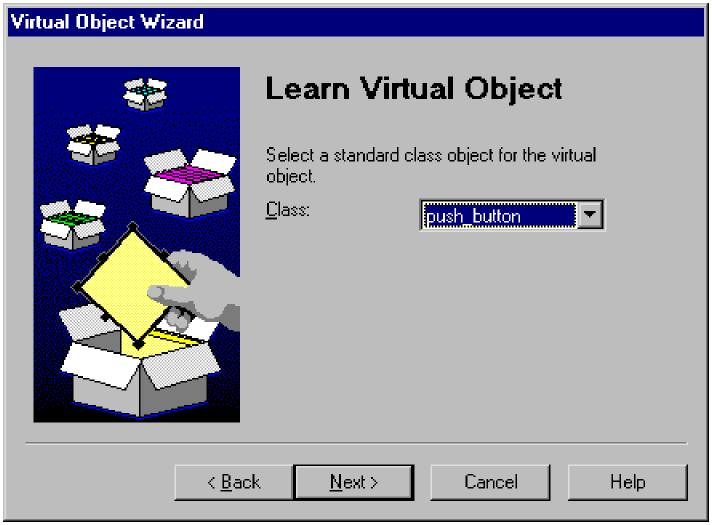
Defining a Virtual Object

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 Virtual Object Wizard on the Tools menu. The Virtual Object wizard opens. Click Next.

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



Select a standard class object for the virtual object.

Class: push_button

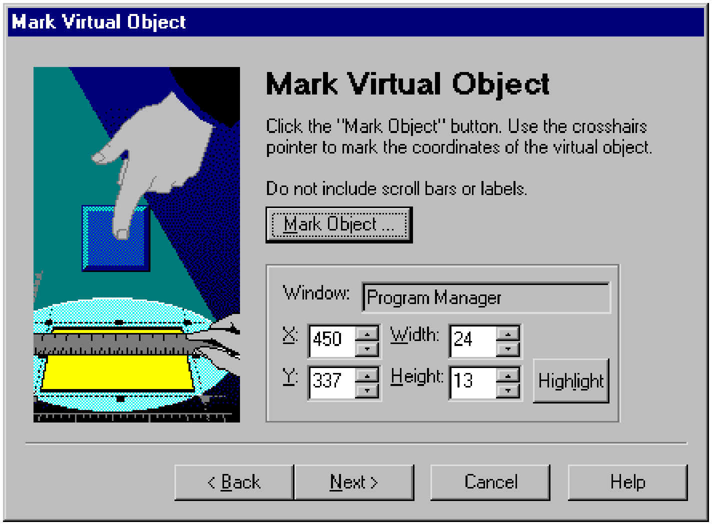
If you select the list class, select the number of visible rows that are displayed in the window. For a table class, select the number of visible rows and columns.

Click Next.

3 Click the Mark Object button. 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 coordinates in the wizard.



Mark Virtual Object

Mark Virtual Object

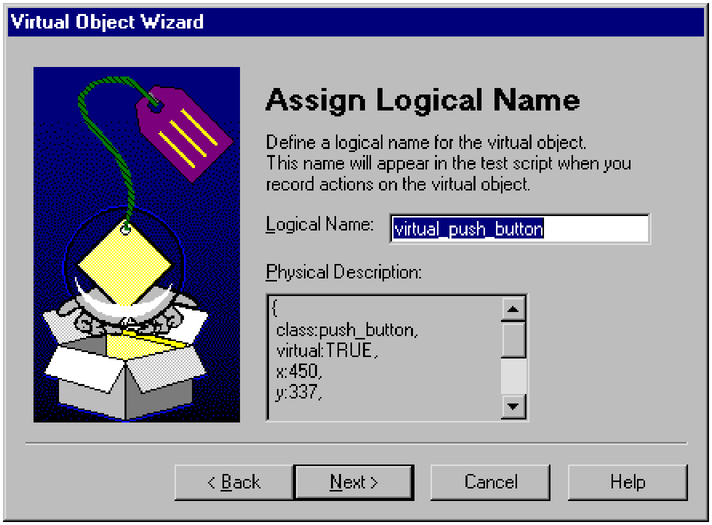
Click the "Mark Object" button. Use the crosshairs pointer to mark the coordinates of the virtual object.

Do not include scroll bars or labels.

Window:	Program Manager
X:	450
Y:	337
Width:	24
Height:	13

If the object marked is visible on the screen, you can click the Highlight button to view it. 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

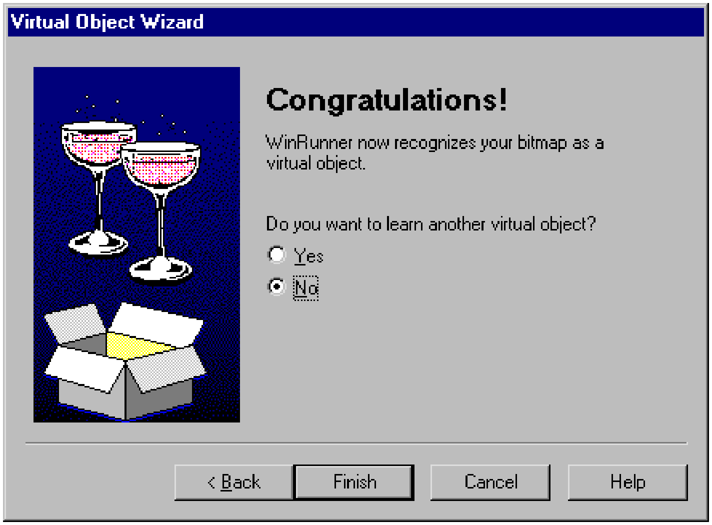
Physical Description:

```
{
 class:push_button,
 virtual:TRUE,
 x:450,
 y:337,
```

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 If you want to learn another virtual object, select Yes. Click Next.

To close the wizard, 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.

Find

?

Creating Tests

Find

?

Creating Tests

Using recording, programming, or a combination of both, you can create automated tests quickly.

This chapter describes:

Find

?

About Creating Tests

You can create tests using both recording and programming. Usually, you start by recording a basic *test script*. As you record, each operation you perform generates a statement in Mercury Interactive's Test Script Language (TSL). These statements appear as a test script in a test window. You can then enhance your recorded test script, either by typing in additional TSL functions and programming elements or by using WinRunner's visual programming tool, the Function Generator.

Two modes are available for recording tests:

- *Context Sensitive* records the operations you perform on your application by identifying Graphical User Interface (GUI) objects.
- *Analog* records keyboard input, mouse clicks, and the precise x- and y-coordinates traveled by the mouse pointer across the screen.

You can further increase the power of your test scripts by adding GUI, bitmap and text checkpoints, as well as synchronization points. Checkpoints allow you to check your application by comparing its current behavior to its behavior in a previous version. Synchronization points solve timing and window location problems that may occur during a test run.

To create a test script, you perform the following main steps:

1 Decide on the functionality you want to test. Determine the checkpoints and synchronization points you need in the test script.

2 Document general information about the test in the Test Properties dialog box.

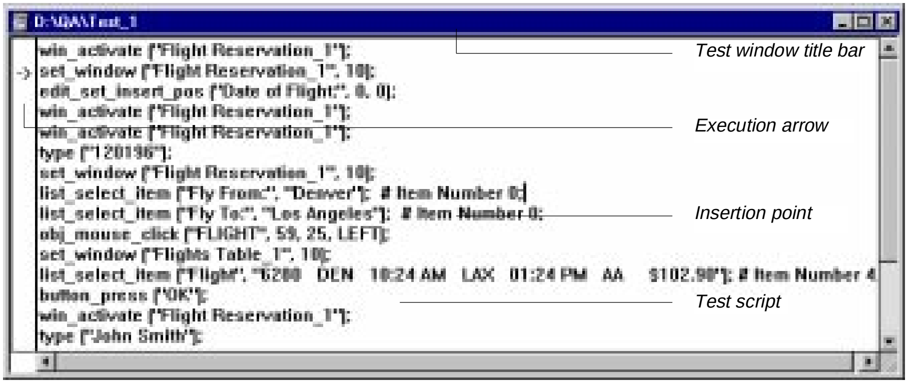
3 Choose a Record mode (*Context Sensitive* or *Analog*) and record the test on your application.

4 Assign a test name and save the test in the file system or in your TestDirector database.

The WinRunner Test Window

You develop and run WinRunner tests in the test window, which contains the following elements:

- *Test window title bar*, which displays the name of the open test
- *Test script*, which consists of statements generated by recording and/or programming in TSL, Mercury Interactive's Test Script Language
- *Execution arrow,* which indicates the line of the test script being executed (to move the marker to any line in the script, click the mouse in the left window margin next to the line)
- *Insertion point,* which indicates where you can insert or edit text



```
  win_activate ("Flight Reservation_1");
->set_window ("Flight Reservation_1", 10);
  edit_set_insert_pos ("Date of Flight:", 0, 0);
  win_activate ("Flight Reservation_1");
  win_activate ("Flight Reservation_1");
  type ("120196");
  set_window ("Flight Reservation_1", 10);
  list_select_item ("Fly From:", "Denver");  # Item Number 0;|
  list_select_item ("Fly To:", "Los Angeles");  # Item Number 0;
  obj_mouse_click ("FLIGHT", 59, 25, LEFT);
  set_window ("Flights Table_1", 10);
  list_select_item ("Flight", "6200   DEN   10:24 AM   LAX   01:24 PM   AA    $102.98"); # Item Number 4
  button_press ("OK");
  win_activate ("Flight Reservation_1");
  type ("John Smith");
```

Context Sensitive Recording

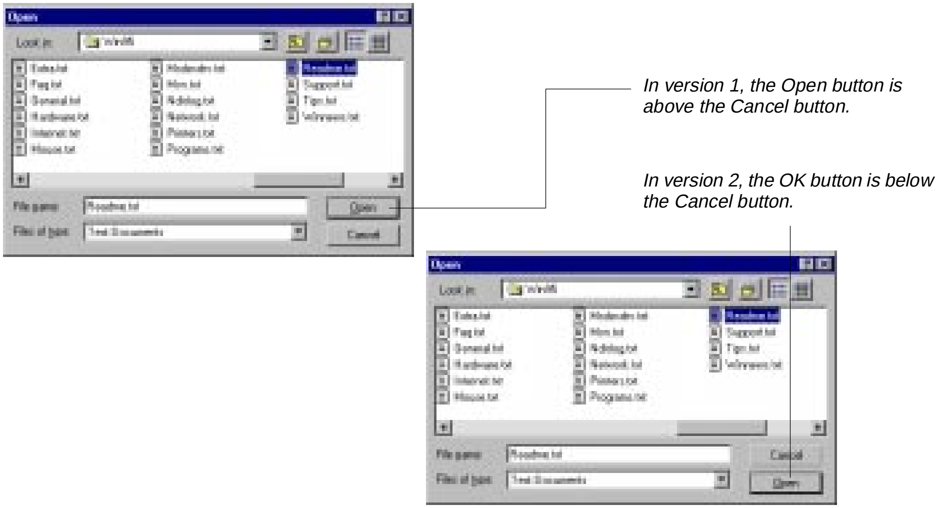
Context Sensitive mode records the operations you perform on your application in terms of its GUI objects. As you record, WinRunner identifies each GUI object you click (such as a window, button, or list), and the type of operation performed (such as drag, click, or select).

For example, if you click the Open button in an Open dialog box, WinRunner records the following:

```
button_press ("Open");
```

When it runs the test, WinRunner looks for the Open dialog box and the Open button represented in the test script. If, in subsequent runs of the test, the button is in a different location in the Open dialog box, WinRunner is still able to find it.

Find
?


In version 1, the Open button is above the Cancel button.
In version 2, the OK button is below the Cancel button.
Open
Look in:
File name:
Files of type:
Open
Cancel

Use Context Sensitive mode to test your application by operating on its user interface. For example, WinRunner can perform GUI operations (such as button clicks and menu or list selections), and then check the outcome by observing the state of different GUI objects (the state of a check box, the contents of a text box, the selected item in a list, etc.).

Remember that Context Sensitive tests work in conjunction with GUI map files. We strongly recommend that you read the “Understanding the GUI Map” section of this guide before you start recording.

Analog Recording

Analog mode records keyboard input, mouse clicks, and the exact path traveled by your mouse. For example, if you choose the Open command from the File menu in your application, WinRunner records the movements of the mouse pointer on the screen. When *WinRunner* executes the test, the mouse pointer retraces the coordinates.

In your test script, the menu selection described above might look like this:

```
# mouse track
move_locator_track (1);

# left mouse button press
mtype ("<T110><kLeft>-");

# mouse track
move_locator_track (2);

# left mouse button release
mtype ("<kLeft>+");
```

Use Analog mode when exact mouse movements are an integral part of the test, such as in a drawing application. Note that you can switch to and from Analog mode during a Context Sensitive recording session.

Note: You cannot run test scripts that were recorded in XRunner in Analog mode in WinRunner. The portions of XRunner test scripts recorded in Analog mode must be rerecorded in WinRunner before running them in WinRunner. For information on configuring GUI maps created in XRunner for WinRunner, see Chapter 6, **Configuring the GUI Map**. For information on using GUI checkpoints created in XRunner in WinRunner test scripts, see Chapter 9, **Checking GUI Objects**. For information on using bitmap checkpoints created in XRunner in WinRunner test scripts, see Chapter 12, **Checking Bitmaps**.

Checkpoints

Checkpoints allow you to compare the current behavior of the application being tested to its behavior in an earlier version.

You can add three types of checkpoints to your test scripts:

- GUI checkpoints verify information about GUI objects. For example, you can check that a button is enabled or see which item is selected in a list. See Chapter 9, **Checking GUI Objects**, for more information.
- Bitmap checkpoints take a “snapshot” of a window or area of your application and compare this to an image captured in an earlier version. See Chapter 12, **Checking Bitmaps**, for more information.
- Text checkpoints read text in GUI objects and in bitmaps and enable you to verify their contents. See Chapter 13, **Checking Text**, for more information.

Synchronization Points

Synchronization points allow you to solve anticipated timing problems between the test and your application. For example, if you create a test that opens a database application, you can add a synchronization point that causes the test to wait until the database records are loaded on the screen.

For Analog testing, you can also use a synchronization point to ensure that WinRunner repositions a window at a specific location. When you run a test, the mouse cursor travels along exact coordinates. Repositioning the window enables the mouse pointer to make contact with the correct elements in the window. See Chapter 14, **Synchronizing Test Execution**, for more information.

Planning a Test

Plan a test carefully before you begin recording or programming. Following are some points to consider:

- Determine the functionality you are about to test. It is better to design short, specialized tests that check specific functions of the application, than long tests that perform multiple tasks.
- Decide on the types of checkpoints and synchronization points you want to use in the test.
- If you plan to use recording, decide which parts of your test should use the Analog recording mode and which parts should to use the Context Sensitive mode.
- Determine the types of programming elements (such as loops, arrays, and user-defined functions) that you want to add to the recorded test script.

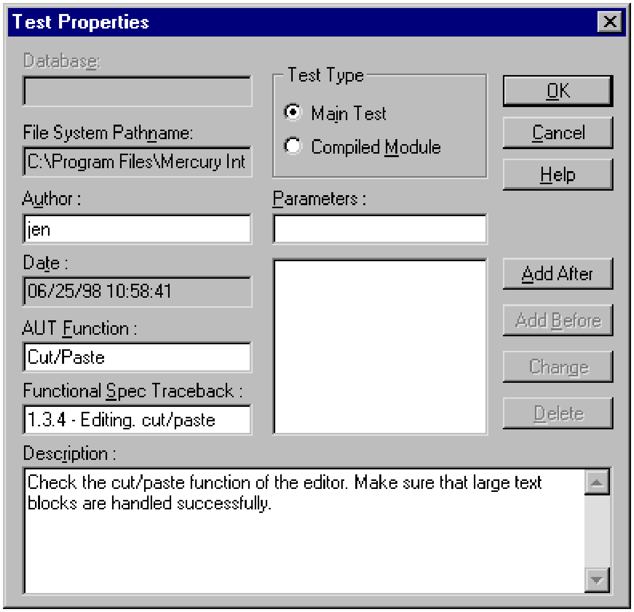
Documenting Test Information

Before creating a test, you should document information about the test in the Test Properties dialog box. You can enter the name of the test author, the type of functionality tested, a detailed description of the test, and a reference to the relevant functional specifications document.

You can also use the Test Properties dialog box to designate a test as a compiled module and to define parameters for a test. For more information, refer to Chapter 20, **Creating Compiled Modules**, and Chapter 18, **Calling Tests**.

To document test information:

1 On the File menu, choose Test Properties to open the Test Properties dialog box.



File System Pathname: C:\Program Files\Mercury Int

Test Type: (● Main Test) ○ Compiled Module

Author : jen

Parameters :

Date : 06/25/98 10:58:41

AUT Function : Cut/Paste

Functional Spec Traceback : 1.3.4 - Editing. cut/paste

Description : Check the cut/paste function of the editor. Make sure that large text blocks are handled successfully.

2 Add information about the test. Note that the Test Properties dialog box automatically displays the current date and time.

3 Click OK to save the test information and close the dialog box.

Recording a Test

Consider the following guidelines when recording a test:

- Before you start to record, close all applications not required for the test.
- Create your test so that it “cleans up” after itself. When the test is completed, the environment should resemble the pre-test conditions. (For example, if the test started with the application window closed, then the test should also close the window and not minimize it to an icon.)
- When recording, use mouse clicks rather than the Tab key to move within a window in the application being tested.
- When recording in Analog mode, use softkeys rather than the WinRunner menus or toolbars to insert checkpoints.
- When recording in Analog mode, avoid typing ahead. For example, when you want to open a window, wait until it is completely redrawn before continuing. In addition, avoid holding down a mouse button when this results in a repeated action (for example, using the scroll bar to move the screen display). Doing so can initiate a time-sensitive operation that cannot be precisely recreated. Instead, use discrete, multiple clicks to achieve the same results.
- WinRunner supports recording and running tests on applications with RTL-style window properties. RTL-style window properties include right-to-left menu order and typing, a left scroll bar, and attached text at the top right corner of GUI objects. WinRunner supports pressing the CTRL and SHIFT keys together or the ALT and SHIFT keys together to change language and direction when typing. The

default setting for attached text supports recording and running tests on applications with RTL-style windows. For more information on attached text options, see Chapter 33, **Setting Global Testing Options**, and Chapter 34, **Setting Testing Options from a Test Script**.

- WinRunner supports recording and running tests on applications with drop-down and menu-like toolbars, which are used in Microsoft Internet Explorer 4.0 and Windows 98. Although menu-like toolbars may look exactly like menus, they are of a different class, and WinRunner records on them differently. When an item is selected from a drop-down or a menu-like toolbar, WinRunner records a **toolbar_select_item** statement. (This function resembles the **menu_select_item** function, which records selecting menu commands on menus.) For more informations, see the *TSL Online Reference*.

To record a test:

1. On the Create menu, choose either Record–Context Sensitive or Record–Analog.
2. Perform the test as planned using the keyboard and mouse.

   Insert checkpoints and synchronization points as needed by choosing the appropriate commands from the floating toolbar or from the Create menu: Check GUI, Check Bitmap, Wait Bitmap, or Get Text.
3. To stop recording, click Stop Recording on the Create menu or click the Stop button.

Activating Test Creation Commands Using Softkeys

You can activate several of WinRunner's commands using softkeys. WinRunner reads input from softkeys even when the WinRunner window is not the active window on your screen, or when it is minimized. Note that you can configure the softkeys. For more information, see Chapter 32, **Customizing WinRunner's User Interface**.

The following table lists the default softkey configurations for test creation:

Command	Default Softkey Combination	Function
STOP	Ctrl Left + F3	Stops test recording.
GET TEXT OBJECT	F11	Captures text in an object or a window.
GET TEXT AREA	Alt Right + F11	Captures text in a specified area.
MOVE LOCATOR	Alt Left + F6	Records a move_locator_abs statement with the current position (in pixels) of the screen pointer.
RECORD	F2	Starts test recording. While recording, this softkey toggles between the Context Sensitive and Analog modes.

Command	Default Softkey Combination	Function
CHECK BITMAP	Ctrl Left + F12	Captures an object or a window bitmap.
CHECK BITMAP AREA	Alt Left + F12	Captures an area bitmap.
WAIT BITMAP	Ctrl Left + F11	Instructs WinRunner to wait for a specific object or window bitmap to appear.
WAIT BITMAP AREA	Alt Left + F11	Instructs WinRunner to wait for a specific area bitmap to appear.
CHECK GUI	Ctrl Right + F12	Checks one GUI object or window.
GUI CHECKLIST	F12	Opens the Check GUI dialog box.
INSERT FUNCTION	F8	Inserts a TSL function for a GUI object.
INSERT FUNCTION FROM LIST	F7	Opens the Function Generator dialog box.

Programming a Test

You can use programming to create an entire test script, or to enhance your recorded tests. WinRunner contains a visual programming tool, the Function Generator, which provides a quick and error-free way to add TSL functions to your test scripts. To generate a function call, simply point to an object in your application or select a function from a list. For more information, see Chapter 17, **Using Visual Programming**.

You can also add general purpose programming features such as variables, control-flow statements, arrays, and user-defined functions to your test scripts. You may type these elements directly into your test scripts. For more information on creating test scripts with programming, see the “Programming with TSL” section of this guide.

Editing a Test

To make changes to a test script, use the commands in the Edit menu or the corresponding toolbar buttons. The following commands are available:

Edit Command	Description
Undo	Cancels the last editing operation.
Cut	Deletes the selected text from the test script and places it onto the Clipboard.
Copy	Makes a copy of the selected text and places it onto the Clipboard.
Paste	Pastes the text on the Clipboard at the insertion point.
Delete	Deletes the selected text.
Select All	Selects all the text in the active test window.
Find	Finds the specified characters in the active test window.
Find Next	Finds the next occurrence of the specified characters.
Find Previous	Finds the previous occurrence of the specified characters.
Replace	Finds and replaces the specified characters with new characters.
Go To	Moves the insertion point to the specified line in the test script.

Managing Test Files

You use the commands in the File menu to create, open, save, print, and close test files.

Creating a New Test

Choose New on the File menu or click the New button. A new window opens, titled *Noname,* and followed by a numeral (for example, *Noname7*). You are ready to start recording or programming a test script.

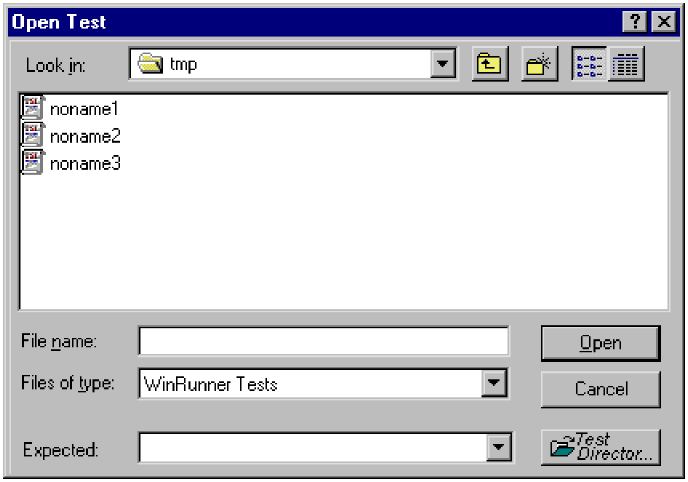
Opening an Existing Test

To open an existing test, choose Open on the File menu or click the Open button.

Note: No more than 100 tests may be open at the same time.

To open a test from the file system:

1 Choose Open on the File menu to open the Open Test dialog box.



2 In the Look In box, click the location of the test you want to open.

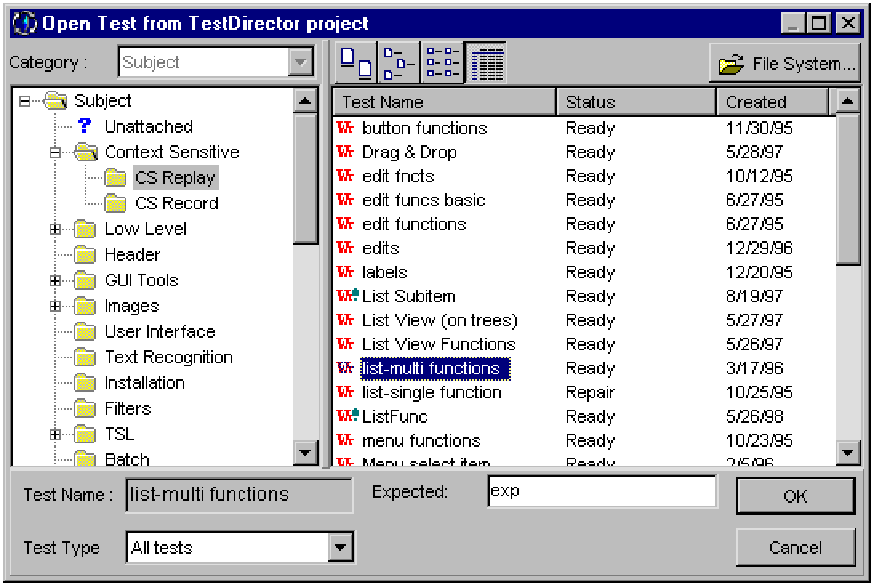
3 In the Test Name box, click the name of the test to open.

4 If the test has more than one set of expected results, click the directory you want to use on the Expected list. The default directory is called *exp.*

5 Click OK to open the test.

To open a test from a TestDirector database:

1 Choose Open on the File menu. The Open Test from TestDirector Database dialog box opens and displays the test plan tree.



Open Test from TestDirector project

Category : Subject

File System...

- Subject
  - Unattached
  - Context Sensitive
    - CS Replay
    - CS Record
  - Low Level
  - Header
  - GUI Tools
  - Images
  - User Interface
  - Text Recognition
  - Installation
  - Filters
  - TSL
  - Batch

Test Name	Status	Created
button functions	Ready	11/30/95
Drag & Drop	Ready	5/28/97
edit fncts	Ready	10/12/95
edit funcs basic	Ready	6/27/95
edit functions	Ready	6/27/95
edits	Ready	12/29/96
labels	Ready	12/20/95
List Subitem	Ready	8/19/97
List View (on trees)	Ready	5/27/97
List View Functions	Ready	5/26/97
list-multi functions	Ready	3/17/96
list-single function	Repair	10/25/95
ListFunc	Ready	5/26/98
menu functions	Ready	10/23/95

Test Name : list-multi functions

Expected: exp

Test Type All tests

OK

Cancel

Note that the Open Test from TestDirector Database dialog box opens only when WinRunner is connected to a TestDirector database.

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 in the Test Name list. The test appears in the read-only Test Name box.

4 If desired, enter an expected results directory for the test in the Expected box. (Otherwise, the default directory is used.)

5 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: You can click the File System button to open the Open Test dialog box and open a test from the file system.

For more information on opening tests in a TestDirector database, see Chapter 37, **Managing the Testing Process**.

Find
?

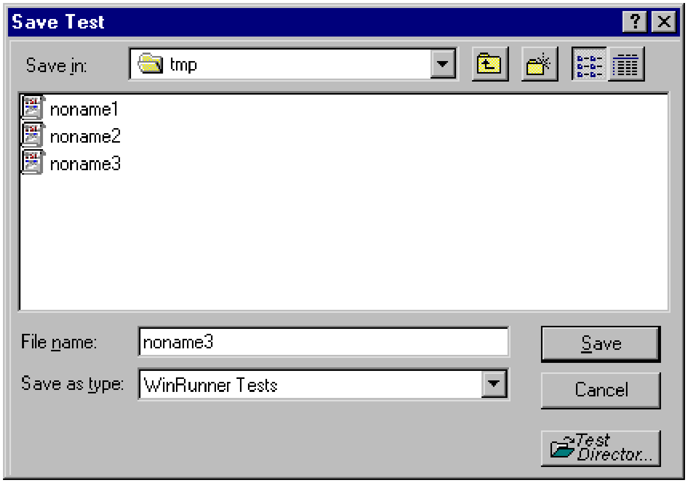
Saving a Test

The following options are available for saving tests:

- Save changes to a previously saved test by choosing Save on the File menu or clicking the Save button.
- Save two or more open tests simultaneously by choosing Save All on the File menu.
- Save a new test script by choosing Save As on the File menu or by clicking the Save button.

To save a test to the file system:

1 Choose a Save command on the File menu, as described above. The Save Test dialog box opens.



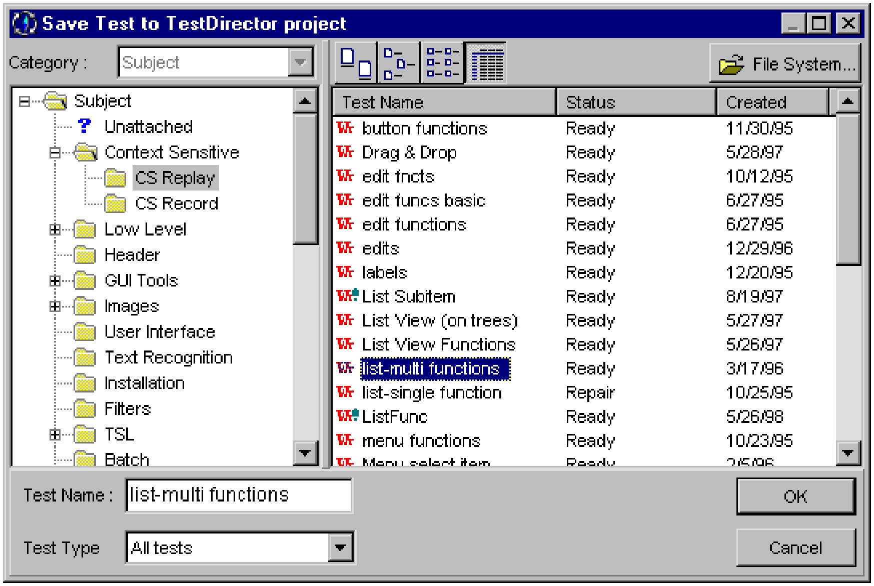
2 In the Look In box, click the location where you want to save the test.

3 Enter the name of the test in the File Name box.

4 Click Save to save the test.

To save a test to a TestDirector database:

1 Choose a Save command on the File menu, as described above. The Save Test to TestDirector Database dialog box opens.



Test Name	Status	Created
button functions	Ready	11/30/95
Drag & Drop	Ready	5/28/97
edit fncts	Ready	10/12/95
edit funcs basic	Ready	6/27/95
edit functions	Ready	6/27/95
edits	Ready	12/29/96
labels	Ready	12/20/95
List Subitem	Ready	8/19/97
List View (on trees)	Ready	5/27/97
List View Functions	Ready	5/26/97
list-multi functions	Ready	3/17/96
list-single function	Repair	10/25/95
ListFunc	Ready	5/26/98
menu functions	Ready	10/23/95

Note that the Save Test to TestDirector Database dialog box opens only when WinRunner is connected to a TestDirector database.

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 text 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: You can click the File System button to open the Save Test dialog box and save a test in the file system.

The next time you start TestDirector, the new test will appear in the TestDirector's test plan tree. Refer to the *TestDirector User's Guide* for more information.

For more information on saving tests to a TestDirector database, see Chapter 37, **Managing the Testing Process**.

Printing a Test

To print a test script, choose Print on the File menu to open the Print dialog box.

- Choose the print options you want.
- Click OK to print.

Closing a Test

- To close the current test, choose Close on the File menu.
- To simultaneously close two or more open tests, choose Close All on the Window menu.

Checking GUI Objects

By adding GUI checkpoints to your test scripts, you can compare the behavior of GUI objects between versions of your application.

This chapter describes:

Find

?

About Checking GUI Objects

Use GUI checkpoints in your test scripts to help you examine GUI objects in your application and detect defects. When you run a test, a GUI *checkpoint* compares the behavior of GUI objects in the current version of the application being tested with the behavior of the same objects in an earlier version.

For example, suppose you want to verify that when a specific dialog box opens, the OK, Cancel, and Help buttons are enabled. You can create a *checkpoint* that captures the state of these buttons, and saves this information as *expected results*. When you run the test, WinRunner compares the current state of these buttons with the expected state captured earlier. If any differences are detected, WinRunner saves this information in the test results.

You can create a GUI *checkpoint* by pointing to GUI objects and choosing the properties that you want WinRunner to check. You can check the default properties recommended by WinRunner, or you can specify which properties to check. Information about the GUI objects and the selected properties is saved in a *checklist*. WinRunner then captures the current property values for the GUI objects and saves this information as *expected results*. A GUI checkpoint is automatically inserted into the test script. This checkpoint appears in your test script as an **obj_check_gui** or **a win_check_gui** statement.

For example, the following statement is generated in the sample Flight application:

```
obj_check_gui ("OK", "list1.ckl", "gui1", 1);
```

where list1.ckl is the name of the *checklist* containing the list of objects. Note that a single checklist may be used in one or more *checkpoints* in a test script.

Note that if you are recording in Analog mode, use the CHECK GUI or the GUI CHECKLIST softkeys. This prevents WinRunner from recording extraneous mouse movements.

When you run the test, WinRunner compares the current state of the GUI objects in the application being tested to the expected results. If the expected results and the current results do not match, the GUI checkpoint fails. The results of the checkpoint can be viewed in the Test Results window. For more information, see Chapter 26, **Analyzing Test Results**.

WinRunner provides special built-in support for various application development environments such as Visual Basic and Power Builder. You can create GUI checkpoints that check the properties of OLE controls and contents, and properties of tables. For more information, see Chapter 10, **Working with ActiveX Controls**, and Chapter 11, **Checking Tables**.

Note that any GUI object you check that is not already in the GUI map is added automatically to the temporary GUI map file. See Chapter 3, **Introducing Context Sensitive Testing**, for more information.

Note: You cannot use GUI checkpoints created in XRunner when you run test scripts in WinRunner. You must recreate the GUI checkpoints in WinRunner. For information on using GUI maps created in XRunner, see Chapter 6, **Configuring the GUI Map**. For information on using test scripts recorded in XRunner in Analog mode, see Chapter 8, **Creating Tests**. For information on using bitmap checkpoints created in XRunner, see Chapter 12, **Checking Bitmaps**.

Checking a Single Object

You can create a GUI checkpoint to check a single object or window in the application being tested. You can either check the object or window with its default properties or specify which properties to check.

Creating a GUI Checkpoint with Default Checks

You can create a GUI checkpoint that checks the default properties recommended by WinRunner. For example, if you create a GUI checkpoint that checks a push button, the default check verifies that the push button is enabled.

To create a GUI checkpoint with default checks:

1 Choose Check GUI > Object/Window on the Create menu, or click the Check GUI - Point button on the Test Creation toolbar. If you are recording in Analog mode, press the CHECK GUI softkey. Note that you can press the CHECK GUI softkey in Context Sensitive mode as well.

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

2 Click an object or a window. If you click a window title bar, a help window prompts you to check all the objects in the window. Click No to check only the window itself.

3 Click OK to close the dialog box.

Find
?

4 WinRunner captures the GUI information and stores it in the test's expected results directory. The WinRunner window is restored and a GUI checkpoint is inserted in the test script as an **obj_check_gui or a win_check_gui** statement. For more information, see **Understanding GUI Checkpoint Statements** on page 155.

Creating a GUI Checkpoint by Specifying which Properties to Check

You can specify which properties to check for an object or a window. For example, if you create a checkpoint that checks a push button, you can choose to verify that it is in focus, instead of enabled.

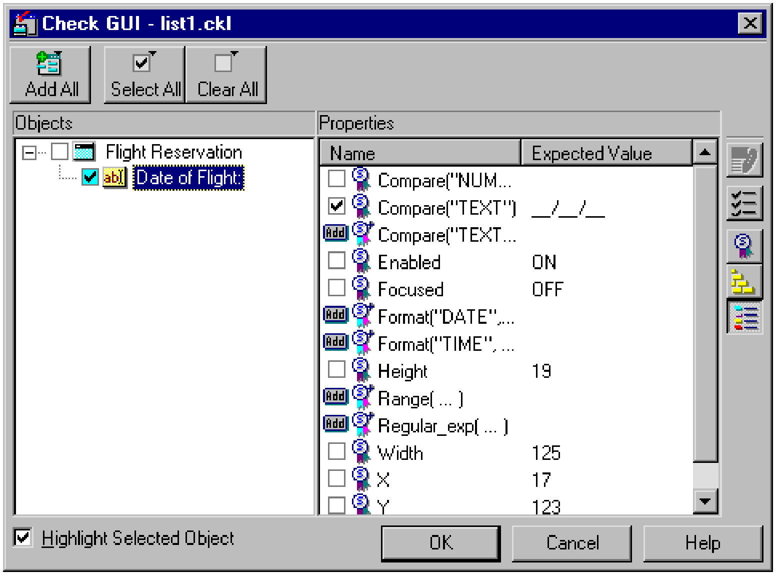
To create a GUI checkpoint by specifying which properties to check

1 Choose Check GUI > Object/Window on the Create menu, or click the Check GUI - Point button on the Test Creation toolbar. If you are recording in Analog mode, press the CHECK GUI softkey.

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

Find
?

2 Double-click the object or window. The Check GUI dialog box opens.



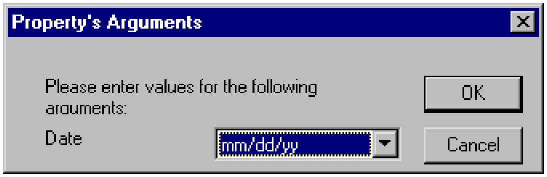
Name	Expected Value
☐ Compare("NUM...	
✓ Compare("TEXT")	__/__/__
Add Compare("TEXT...	
☐ Enabled	ON
☐ Focused	OFF
Add Format("DATE",...	
Add Format("TIME", ...	
☐ Height	19
Add Range(...)	
Add Regular_exp(...)	
☐ Width	125
☐ X	17
☐ Y	123

3 Click an object name in the Objects column. The Properties column lists all the properties for the selected object. Select the properties you want to check.

- To edit the expected value of a property, select a property and click the Edit Expected Value button, or double-click the expected value that you want to edit.

- To add a check in which you specify an argument, click the Add button next to a property name, or double-click a property name with an Add button next to it. The Property’s Arguments dialog box opens, and enables you to choose the arguments for the specified property. Once you have added Arguments to a property, the property name appears indented in the list, with a selected check box beside it.

  For example, if you click the Add button next to Date Format property, the following dialog box opens:



Property's Arguments

Please enter values for the following arguments:

Date: mm/dd/yy

  Choose a date format and click OK to close the dialog box. For more information on adding checks with arguments, see **Edit Class** on page 185.

4 Click OK to close the dialog box.

WinRunner captures the GUI information and stores it in the test’s expected results directory. The WinRunner window is restored and a GUI checkpoint is inserted in the test script as an **obj_check_gui or a win_check_gui** statement. For more information, see **Understanding GUI Checkpoint Statements** on page 155.

For more information on the Check GUI dialog box, see **Understanding the GUI Checkpoint Dialog Boxes** on page 166.

Checking Two or More Objects in a Window

You can use a GUI checkpoint to check two or more objects in a window.

To create a GUI Checkpoint for two or more objects:

1 Choose Check GUI > Create GUI Checkpoint on the Create menu. If you are recording in Analog mode, press the GUI CHECKLIST softkey. The Create GUI Checkpoint dialog box opens.

2 Click the Add button. The mouse pointer becomes a pointing hand and a help window opens.

3 To add an object, click it once. If you click a window title bar or menu bar, a help window prompts you to check all the objects in the window. Click No to check only the window object. Click Yes to check the window object and all the objects within it.

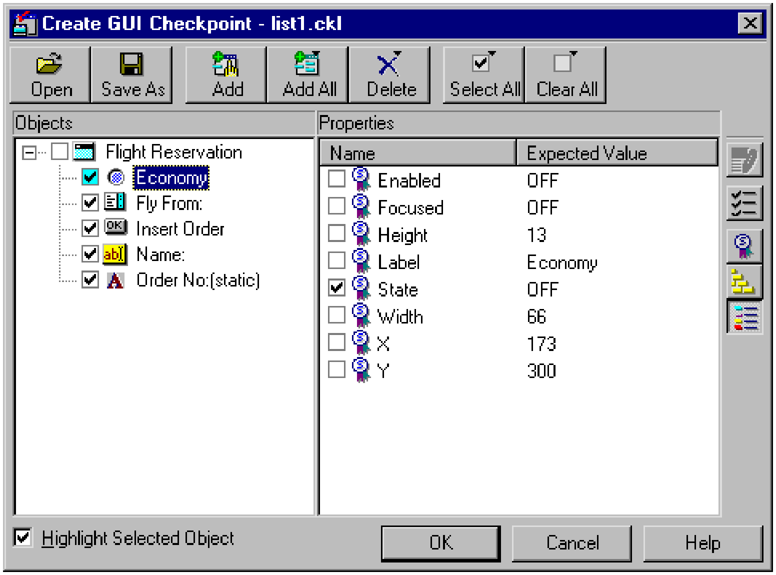
4 The pointing hand remains active. You can continue to choose objects by repeating step 3 above for each object you want to check.

Note: You cannot insert objects from different windows into a single checkpoint.

---

5 Click the right mouse button to stop the selection process and to restore the mouse pointer to its original shape.

The Create GUI Checkpoint dialog box reopens. The Objects column contains the name of the window and objects included in the GUI checkpoint. The Properties column lists all the properties of a selected object.



Objects:
- [ ] Flight Reservation
  - [x] Economy
  - [x] Fly From:
  - [x] Insert Order
  - [x] Name:
  - [x] Order No:(static)

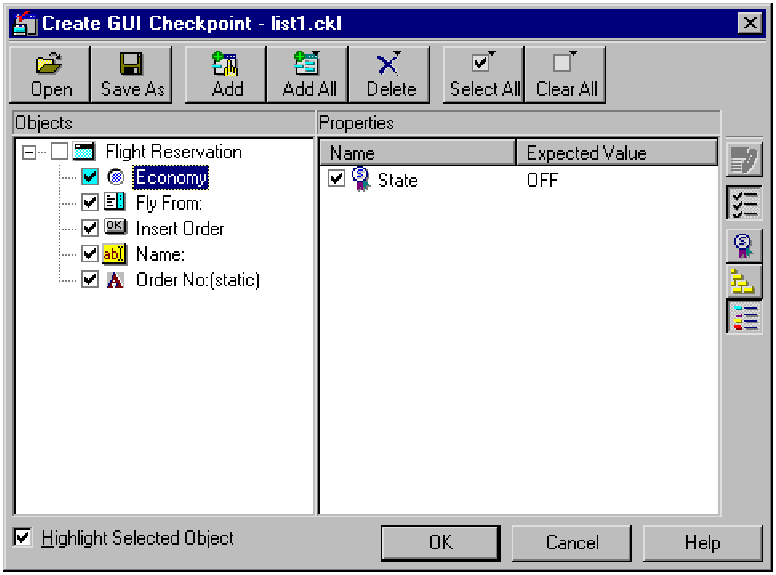
Name	Expected Value
[ ] Enabled	OFF
[ ] Focused	OFF
[ ] Height	13
[ ] Label	Economy
[x] State	OFF
[ ] Width	66
[ ] X	173
[ ] Y	300

[x] Highlight Selected Object

6 To specify which objects to check, click the object name in the Objects column.

The Properties column lists all the properties of the object. The default properties are selected.

- By default, all properties of the selected object are displayed. To view only the selected properties of the specified object, click the Show Selected Properties Only button. This button toggles between a view of all properties and a view of the selected properties only.

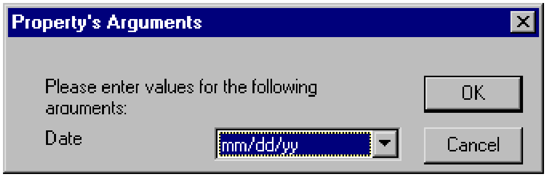


Select any other properties that you want to check.

- To edit the expected value of a property, select a property and click the Edit Expected Value button.

- To add a check in which you specify the argument, click the Add button next to a property name, or double-click a property name with an Add button next to it. The Property’s Arguments dialog box opens, and enables you to choose the arguments for the specified property. Once you have added Arguments to a property, the property name appears indented in the list, with a selected check box beside it.

  For example, if you click the Add button next to Date Format property, the following dialog box opens:



Please enter values for the following arguments:

Date mm/dd/yy

  Choose a date format and click OK to close the dialog box. For more information on adding checks with arguments, see **Edit Class** on page 185.

7 To save the checklist and close the dialog box, click OK.

  WinRunner captures the GUI information and stores it in the expected results directory. A **win_check_gui** statement is inserted in the test script. For more information, see **Understanding GUI Checkpoint Statements** on page 155.

  For more information on the Create GUI Checkpoint dialog box, see **Understanding the GUI Checkpoint Dialog Boxes** on page 166.

Checking All Objects in a Window

You can use a GUI checkpoint to check all GUI objects in a window.

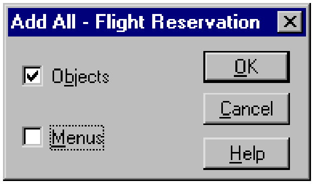
To create a GUI checkpoint that checks all GUI objects in a window:

1 Choose Check GUI > Object/Window on the Create menu, or click the Check GUI - Point button on the Test Creation toolbar. If you are recording in Analog mode, press the CHECK GUI OBJECT/WINDOW softkey. Note that you can press the CHECK GUI OBJECT/WINDOW softkey in Context Sensitive mode as well.

The WinRunner window is minimized to an icon, the mouse pointer turns into a pointing hand, and a help window opens.

2 Click once on the title bar or menu bar of the window you want to check. The WinRunner Message dialog box prompts you to check all the objects in the window. Click Yes.

3 The Add All dialog box opens.



Select Objects, Menus, or both to indicate the types of objects to include in the checklist. When you select only Objects (the default setting), all objects in the window except for menus are included in the checklist. To include menu objects in the checklist, select Menus.

4 Click OK to close the dialog box. WinRunner generates a new checklist containing the objects specified. This may take a few seconds.

WinRunner captures the GUI information and stores it in the test's expected results directory. The WinRunner window is restored and a **win_check_gui** statement is inserted in the test script.

Understanding GUI Checkpoint Statements

A GUI checkpoint appears in your test script as an **obj_check_gui or a win_check_gui** statement.

A *checklist* lists the objects and properties that need to be checked. A *checkpoint* combines a checklist and a list of expected results (saved in the expected results directory, specified when you open the test). The same checklist can be used in several checkpoints to repeat the same set of checks. Multiple expected results directories can be used to test for different values of properties in different scenarios.

For an object, a GUI checkpoint has the following syntax:

obj_check_gui (*object, checklist, GUI_file, time*);

The *object* is the logical name of the GUI object. The *checklist* is the name of the checklist defining the objects and properties to check. The *GUI_file* is the name of the file that stores the expected GUI data. The *time* is the interval marking the maximum delay between the previous input event and the capture of the current GUI data, in seconds. This interval is added to the timeout test option during test execution.

For example, if you click the OK button in the Login window in the Flight application, the resulting statement might be:

```
obj_check_gui ("OK", "list1.ckl", "gui1", 1);
```

For a window, the syntax is:

win_check_gui (*window, checklist, GUI_file, time*);

The *window* is the logical name of the GUI object. The *checklist* is the name of the checklist defining the GUI checks. The *GUI_file* is the name of the file that stores the expected GUI data. The *time* is the interval marking the maximum delay between the previous input event and the capture of the current GUI data, in seconds. This interval is added to the timeout test option during test execution.

For example, if you click the title bar of the Login window in the sample Flight application, the resulting statement might be:

```
win_check_gui ("Login", "list1.ckl", "gui1", 1);
```

Note that WinRunner names the first checklist in the test *list1.ckl* and the first expected results file *gui1*. For more information on the **obj_check_gui** and **win_check_gui** functions, refer to the *TSL Online Reference*.

Modifying GUI Checklists

You can make changes to a checklist you created for a GUI checkpoint.

You can:

- edit a checklist
- use a previously created GUI checklist in a new GUI checkpoint
- make a checklist available to other users

Editing Checklists

You can edit an existing GUI checklist. Note that before you start working, the objects in the checklist must be loaded into the GUI map.

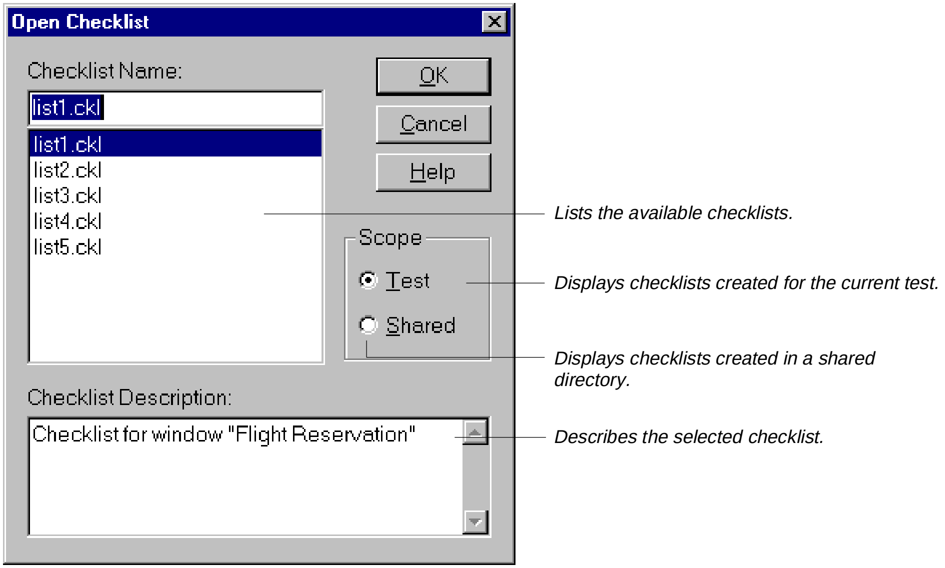
To edit an existing GUI checklist:

1 Choose Check GUI > Edit GUI Checklist on the Create menu. The Open Checklist dialog box opens.

2 A list of checklists for the current test is displayed. To see checklists in a shared directory, click Shared.

Find
?

For more information, see **Saving a GUI Checklist in a Shared Directory** on page 165.



Checklist Name: list1.ckl

list1.ckl
list2.ckl
list3.ckl
list4.ckl
list5.ckl

Scope: (● Test) ○ Shared

Checklist Description: Checklist for window "Flight Reservation"

Lists the available checklists.

Displays checklists created for the current test.

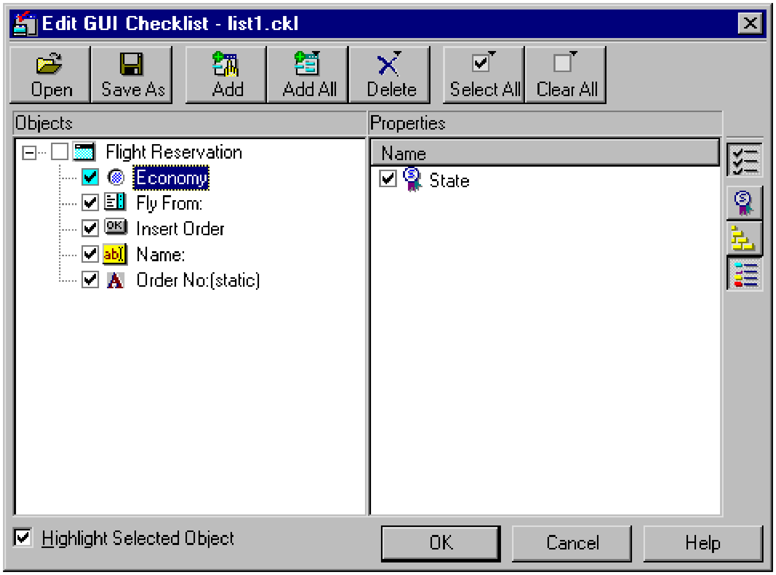
Displays checklists created in a shared directory.

Describes the selected checklist.

3 Select a checklist.

4 Click OK.

The Open Checklist dialog box closes. The Edit GUI Checklist dialog box displays the selected checklist.



5 To see a list of the properties to check for a specific object, click the object name in the Objects column. The Properties column lists all the properties for the selected object.

- To change the properties of an object, click the object name in the Objects column. Select the desired properties from the Properties column.

Find
?

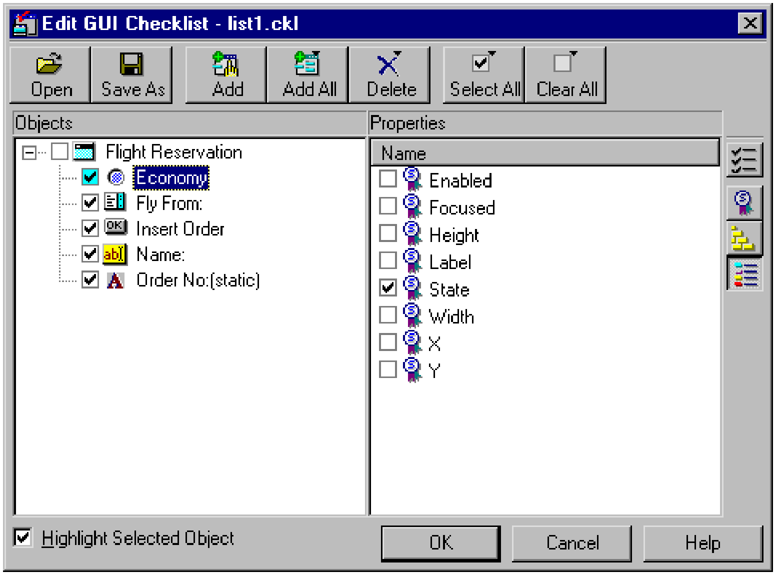
- To delete an object from the checklist, click the object name in the Objects column. Click the Delete button and then select the Object option.
- To add an object to the checklist, make sure the relevant window is open in the application being tested. Click the Add button. The mouse pointer becomes a pointing hand and a help window opens.

  Click each object that you want to include in your checklist. Click the right mouse button to stop the selection process. The Edit GUI Checklist dialog box reopens.

  Note: You cannot insert objects from different windows into a single checklist.

  In the Properties column, select the properties you want to check or accept the default checks.
- To add all objects or menus to the checklist, make sure the window of the application you are testing is active. Click the Add All button and select Objects or Menus.

- By default, all properties of the selected object are displayed. To view only the selected properties of the specified object, click the Show Selected Properties Only button. This button toggles between a view of all properties and a view of the selected properties only.

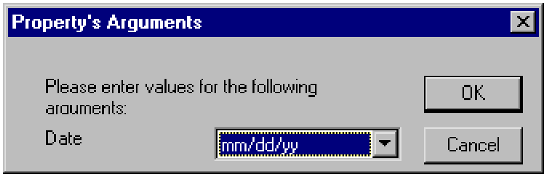


Select any other properties that you want to check.

- To edit the expected value of a property, select a property and click the Edit Expected Value button.

- To add a check in which you specify arguments, click the Add button next to a property name, or double-click a property name with an Add button next to it. The Property's Arguments dialog box opens, and enables you to choose the arguments for the specified property. Once you have added Arguments to a property, the property name appears indented in the list, with a selected check box beside it.

  For example, if you click the Add button next to Date Format property, the following dialog box opens:



Please enter values for the following arguments:

Date mm/dd/yy

  Choose a date format and click OK to close the dialog box. For more information on adding checks with arguments, see **Edit Class** on page 185.

6 Click OK to save the changes in the Edit GUI Checklist dialog box. A help window prompts you to overwrite the existing checklist. Click OK.

  Note that when you overwrite a checklist, any expected results captured earlier remain unchanged. To update the expected results to match the changes in the checklist, you must run the test in Update run mode before you run it in Verify run mode.

7 To save the checklist, click the Save As button. The Save Checklist dialog box opens. Type a new name or use the default name. Click OK. Note that if you do not click the Save As button, WinRunner automatically saves the checklist under its current name when you click OK to close the Edit GUI Checklist dialog box.

The WinRunner window is restored. A new GUI checkpoint statement is not inserted in your test script.

For more information on the Edit GUI Checklist dialog box, see **Understanding the GUI Checkpoint Dialog Boxes** on page 166.

Note: Before you run your test in Verify run mode, you must update the expected results to match the changes you made in the checklist. To update the expected results, run your test in Update run mode. For more information on running a test in Update run mode, see **WinRunner Test Execution Modes** on page 454.

Find

?

Using an Existing GUI Checklist in a Test

The following steps describe how to create a GUI checkpoint using an existing GUI checklist. Note that the appropriate GUI map file must be loaded before you run the test. This ensures that WinRunner can locate the objects to check in your application.

To use an existing checklist in a test:

1 Choose Check GUI > Create GUI Checkpoint on the Create menu. The Create GUI Checkpoint dialog box opens.

2 Click the Open button. The Open Checklist dialog box opens. To see checklists in the Shared directory, select Shared. Select a checklist and click OK. The Open Checklist dialog box closes and the selected list appears in the Create GUI Checkpoint dialog box.

3 Open the window in the application being tested that contains the objects shown in the checklist (if it is not already open).

4 Click OK. WinRunner captures the GUI information and a **win_check_gui statement** is inserted into your test script.

Saving a GUI Checklist in a Shared Directory

By default, checklists for GUI checkpoints are stored in the directory of the current test. You can specify that a checklist be placed in a shared directory to enable wider access.

The default directory in which WinRunner stores your shared checklists is *WinRunner installation directory/*chklist. To choose a different directory, you can use the Shared Checklists box in the Folders tab of the Options dialog box. For more information, see Chapter 33, **Setting Global Testing Options**.

To save a GUI checklist in a shared directory:

1 Choose Check GUI > Edit GUI Checklist on the Create menu. The Open Checklist dialog box opens.

2 Select a checklist and click OK.

The Open Checklist dialog box closes. The Edit GUI Checklist dialog box displays the selected checklist.

3 Save the checklist by clicking Save As. The Save Checklist dialog box opens.

Under Scope, click Shared. Type in a name for the shared checklist. Click OK to save the checklist and to close the dialog box.

4 Click OK to close the Edit GUI Checklist dialog box.

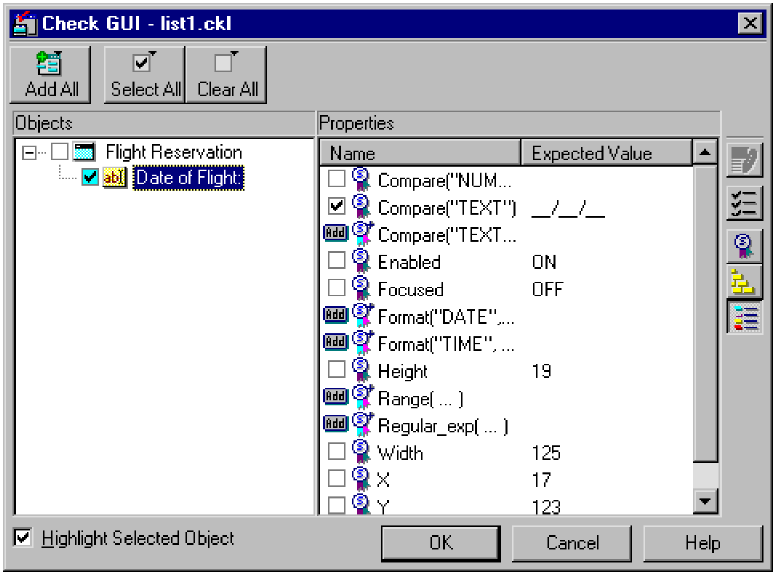
Understanding the GUI Checkpoint Dialog Boxes

When creating a GUI checkpoint to check your GUI objects, you can check the default properties, specify the properties to check, create checklists, and modify checklists. The following are the dialog boxes used to create and maintain your GUI checkpoints:

- The *Check GUI* dialog box enables you to create a GUI checkpoint by specifying which properties to check for a single object or window.
- The *Create GUI Checkpoint* dialog box enables you to create a GUI checklist by accepting the default checks for multiple objects or by selecting which of their properties to check.
- The *Edit GUI Checklist* dialog box enables you to modify the checklist.

The Check GUI Dialog Box

You can use the Check GUI dialog box to create a GUI checkpoint with the checks you specify for a single object or a window. When you choose Create > Check GUI > Object/Window and double-click object or window, the Check GUI dialog box opens.



Check GUI - list1.ckl

Add All | Select All | Clear All

Objects
- [ ] Flight Reservation
  - [✓] Date of Flight:

Properties

	Name	Expected Value
☐	Compare("NUM...	
✓	Compare("TEXT")	__/__/__
Add	Compare("TEXT...	
☐	Enabled	ON
☐	Focused	OFF
Add	Format("DATE",...	
Add	Format("TIME", ...	
☐	Height	19
Add	Range(...)	
Add	Regular_exp(...)	
☐	Width	125
☐	X	17
☐	Y	123

[✓] Highlight Selected Object

OK | Cancel | Help

The Objects column contains the name of the window and objects that will be included in the GUI checkpoint. The Properties column lists all the properties of a

Find
?

selected object. A check mark indicates that the item is selected and is included in the checkpoint.

Clicking the Add button adjacent to some of the properties opens the Property's Arguments dialog box, which prompts the user to enter values for arguments for the specified property.

When you select an object from the Objects column, the Highlight Selected Object option highlights the actual GUI object if the object is visible on the screen.

The Check GUI dialog box includes the following options:

- The Add All button adds all objects or menus in a window to your checklist.
- The Select All button selects all objects, properties, or objects of a given class in the Check GUI dialog box.
- The Clear All button clears all objects, properties, or objects of a given class in the Check GUI dialog box.

This button enables you to edit the expected value of a selected property.

The following buttons enable you to determine the types of properties displayed in the Properties column:

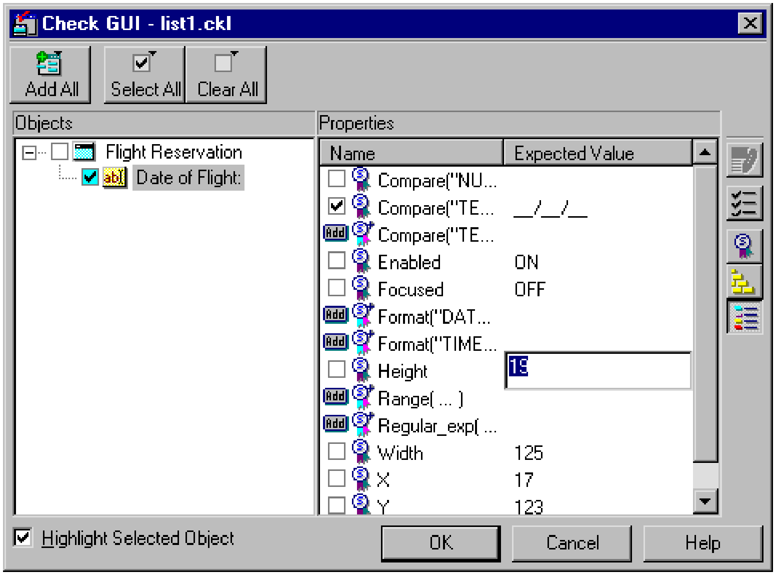
Displays selected properties only. (Toggles between viewing all properties and viewing selected properties only.)

Displays standard properties only.

Displays nonstandard properties only.

Displays all properties.

- To edit the expected value of a property, select a property and click the Edit Expected Value button.



Check GUI - list1.ckl

Add All | Select All | Clear All

Objects

- Flight Reservation
  - ✓ Date of Flight:

Properties

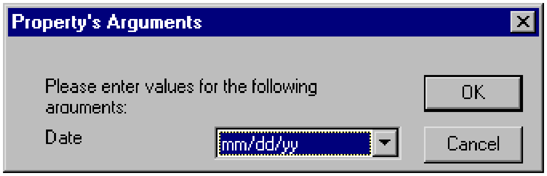
	Name	Expected Value
☐	Compare("NU...	
✓	Compare("TE...	__/__/__
Add	Compare("TE...	
☐	Enabled	ON
☐	Focused	OFF
Add	Format("DAT...	
Add	Format("TIME...	
☐	Height	19
Add	Range(...)	
Add	Regular_exp(...	
☐	Width	125
☐	X	17
☐	Y	123

✓ Highlight Selected Object

OK | Cancel | Help

- To add a check in which you specify the argument, click the Add button next to a property name, or double-click a property name with an Add button next to it. The Property’s Arguments dialog box opens, and enables you to choose the arguments for the specified property. Once you have added Arguments to a property, the property name appears indented in the list, with a selected check box beside it.

  For example, if you click the Add button next to Date Format property, the following dialog box opens:



Please enter values for the following arguments:

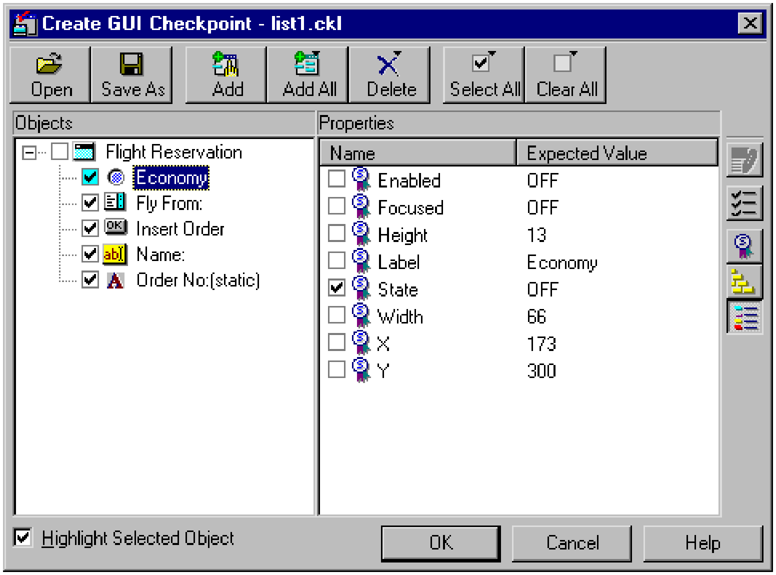
Date mm/dd/yy

Choose a date format and click OK to close the dialog box. For more information on adding checks with arguments, see **Edit Class** on page 185.

When you click OK to close the dialog box, WinRunner captures the GUI information and stores it in the test’s expected results directory. The WinRunner window is restored and a GUI checkpoint is inserted in the test script as an **obj_check_gui or a win_check_gui** statement.

The Create GUI Checkpoint Dialog Box

You can use the Create GUI Checkpoint dialog box to create a GUI checklist with default checks for multiple objects or by specifying which properties to check. To open the Create GUI Checkpoint dialog box, choose Check GUI > Create GUI Checkpoint on the Create menu or click the Check GUI - Create button on the Test Creation toolbar.



Name	Expected Value
☐ Enabled	OFF
☐ Focused	OFF
☐ Height	13
☐ Label	Economy
✓ State	OFF
☐ Width	66
☐ X	173
☐ Y	300

The Objects column contains the name of the window and objects that will be included in the GUI checkpoint. The Properties column lists all the properties of a selected object. A check mark indicates that the item is selected and is included in the checkpoint.

Clicking the Add button adjacent to some of the properties opens the Property's Arguments dialog box, which prompts the user to enter values for arguments for the specified property.

When you select an object from the Objects column, the Highlight Selected Object option highlights the actual GUI object if the object is visible on the screen.

The Create GUI Checkpoint dialog box includes the following options:

- The Open button retrieves a GUI checklist.
- The Save As button saves your GUI checklist. Note that if you do not click the Save As button, WinRunner automatically saves the checklist under its existing name when you click OK to close the Create GUI Checkpoint dialog box.
- The Add button adds an object to your GUI checkpoint.
- The Add All button adds all objects or menus in a window to your GUI checkpoint.
- The Delete button deletes an object, or all of the objects that appear in the GUI checkpoint.
- The Select All button selects all objects, properties, or objects of a given class in the Create GUI Checkpoint dialog box.

Find

?

- The Clear All button clears all objects, properties, or objects of a given class in the Create GUI Checkpoint dialog box.

This button enables you to edit the expected value of a selected property.

The following buttons enable you to determine the types of properties displayed in the Properties column:

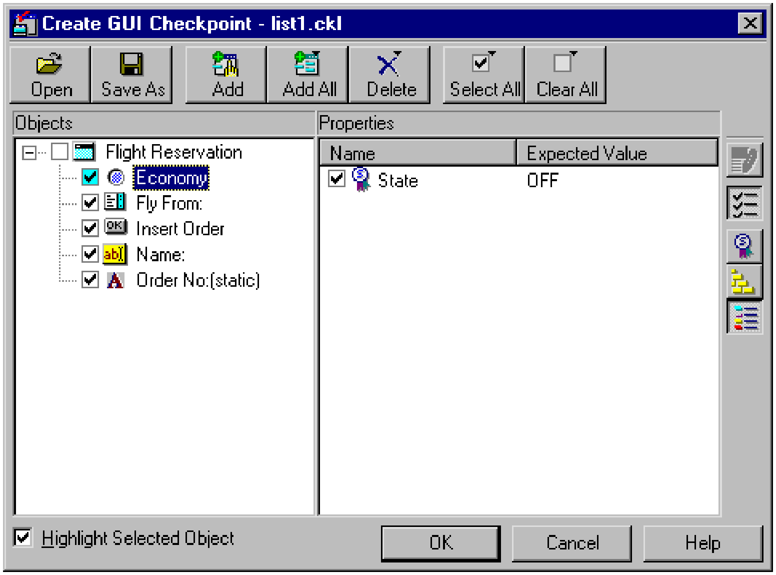
Displays selected properties only. (Toggles between viewing all properties and viewing selected properties only.)

Displays standard properties only.

Displays nonstandard properties only.

Displays all properties.

- To view all properties of the selected object, click the Show Selected Properties Only button.



Objects:
- Flight Reservation
  - ✓ Economy
  - ✓ Fly From:
  - ✓ Insert Order
  - ✓ Name:
  - ✓ Order No:(static)

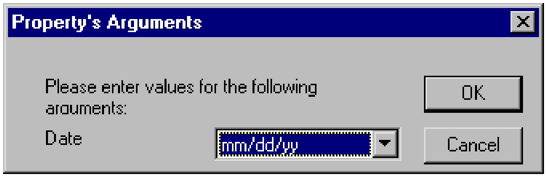
Name	Expected Value
✓ State	OFF

Select any other properties that you want to check.

- To edit the expected value of a property, select a property and click the Edit Expected Value button.

- To add a check in which you specify the argument, click the Add button next to a property name, or double-click a property name with an Add button next to it. The Property's Arguments dialog box opens, and enables you to choose the arguments for the specified property. Once you have added Arguments to a property, the property name appears indented in the list, with a selected check box beside it.

  For example, if you click the Add button next to Date Format property, the following dialog box opens:


Property's Arguments

Please enter values for the following arguments:

Date mm/dd/yy

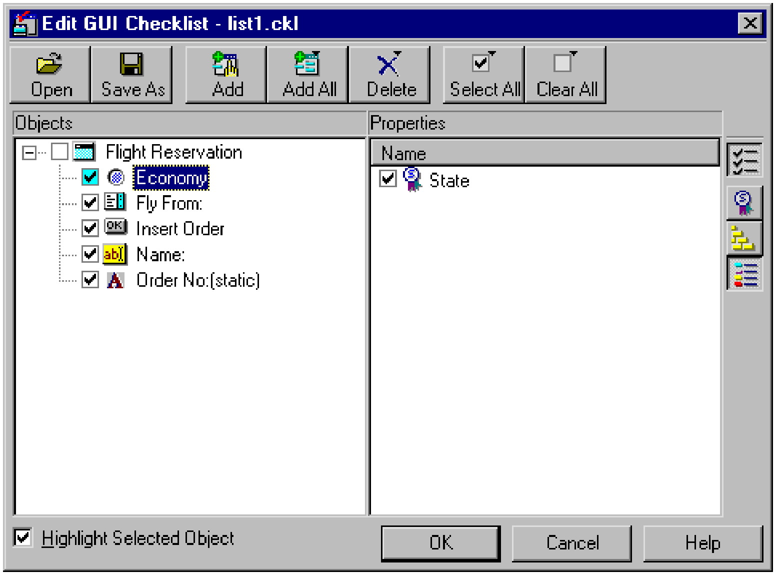
Choose a date format and click OK to close the dialog box. For more information on adding checks with arguments, see **Edit Class** on page 185.

When you click OK to close the dialog box, WinRunner saves your changes, captures the GUI information, and stores it in the test's expected results directory. The WinRunner window is restored and a GUI checkpoint is inserted in the test script as a **win_check_gui** statement.

The Edit GUI Checklist Dialog Box

You can use the Edit GUI Checklist dialog box to modify your checklist. Since a checklist contains only a list of objects and properties, you cannot edit the expected values of an object’s properties in this dialog box.

To open the Edit GUI Checklist dialog box, choose Check GUI > Edit GUI Checklist from the Create menu.



Flight Reservation
- ✓ Economy
- ✓ Fly From:
- ✓ Insert Order
- ✓ Name:
- ✓ Order No:(static)

✓ State

✓ Highlight Selected Object

The Objects column contains the name of the window and objects that are included in the checklist. The Properties column lists all the properties for a selected object. A check mark indicates that the item is selected and will be checked in checkpoints that use this checklist.

Clicking the Add button next to some of the properties opens the Property's Arguments dialog box, which prompts the user to enter values for arguments for the specified property.

When you select an object from the Objects column, the Highlight Selected Object option highlights the actual GUI object if the object is visible on the screen.

The Edit GUI Checklist dialog box includes the following options:

- The Open button retrieves a GUI checklist.
- The Save As button saves your GUI checklist. Note that if you do not click the Save As button, WinRunner will automatically save the checklist under its existing name when you click OK to close the Edit GUI Checklist dialog box.
- The Add button adds an object to your GUI checklist.
- The Add All button adds all objects or menus in a window to your GUI checklist.
- The Delete button deletes an object, or all of the objects that appear in the GUI checklist.
- The Select All button selects all objects, properties, or objects of a given class in the Edit GUI Checklist dialog box.

- The Clear All button clears all objects, properties, or objects of a given class in the Edit GUI Checklist dialog box.

  The following buttons enable you to determine the types of properties displayed in the Properties column:

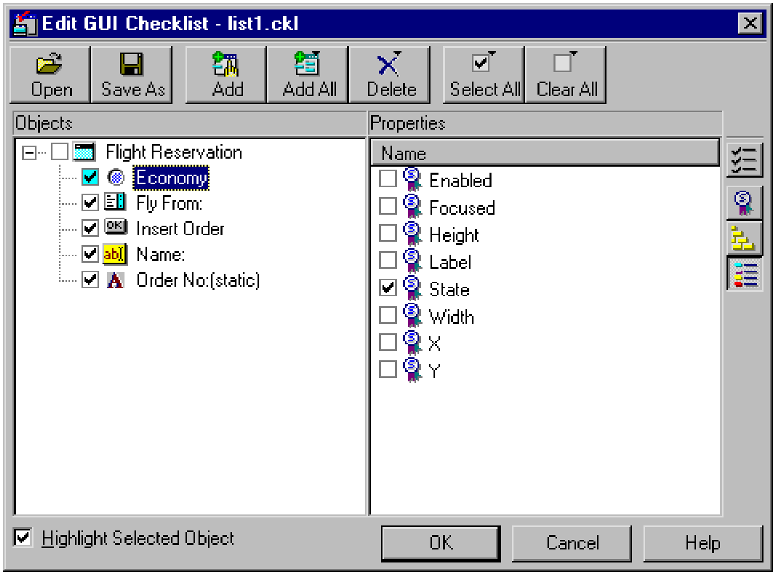
Displays selected properties only. (Toggles between viewing all properties and viewing selected properties only.)

Displays standard properties only.

Displays nonstandard properties only.

Displays all properties.

- To view all properties of the selected object, click the Show Selected Properties Only button.



Objects:
- ☐ Flight Reservation
  - ☑ Economy
  - ☑ Fly From:
  - ☑ Insert Order
  - ☑ Name:
  - ☑ Order No:(static)

Properties:

Name
☐ Enabled
☐ Focused
☐ Height
☐ Label
☑ State
☐ Width
☐ X
☐ Y

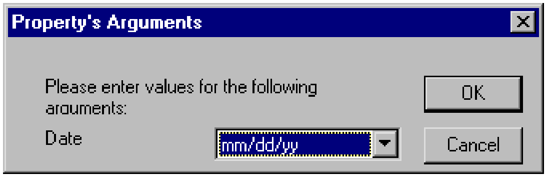
☑ Highlight Selected Object

Select any other properties that you want to check.

- To edit the expected value of a property, select a property and click the Edit Expected Value button.

- To add a check in which you specify the argument, click the Add button next to a property name, or double-click a property name with an Add button next to it. The Property's Arguments dialog box opens, and enables you to choose the arguments for the specified property. Once you have added Arguments to a property, the property name appears indented in the list, with a selected check box beside it.

  For example, if you click the Add button next to Date Format property, the following dialog box opens:



Please enter values for the following arguments:

Date mm/dd/yy

Choose a date format and click OK to close the dialog box. For more information on adding checks with arguments, see **Edit Class** on page 185.

When you click OK to close the dialog box, WinRunner prompts you to overwrite your checklist. Note that when you overwrite a checklist, any expected results captured earlier in checkpoints using the edited checklist remain unchanged.

The WinRunner window is restored. A new GUI checkpoint statement is not inserted in your test script.

Note: Before your run your test in Verify run mode, you must update the expected results to match the changes you made in the checklist. To update the expected results, run your test in Update run mode. For more information on running a test in Update run mode, see **WinRunner Test Execution Modes** on page 454.

Find

?

Checking Properties Using check_info Functions

You can check the property values of GUI objects using the **check_info** TSL functions. A check function is available for each standard object class (for example, **button_check_info**, **edit_check_info**, **menu_check_info**).

For example, **button_check_info** has the following syntax:

button_check_info (*button, property*, *property_value*);

The button is the name of the button. The *property* is the property to check. The *property_value* is a variable that stores the current value of the property.

In most cases you use a **check_info** function together with a **get_info** function. The **get_info** function gets the current value of an property; the **check_info** function checks that this property value is correct.

For example, suppose that in the Flight Reservation application you want to check the value displayed in the “Total” box. This value should be equal to the number in the “Tickets” box multiplied by the price per ticket in the “Price” box. In the test segment below, **edit_get_info** gets the value in the “Tickets” box and assigns it to the variable “T”. This function also gets the value in the “Price” box and assigns it to the variable “P”. Then **edit_check_info** checks that the value in the “Total” box is equal to P multiplied by T. If this value is incorrect, a message is sent to the report.

```
edit_get_info("Tickets:","value",T);
edit_get_info("Price:","value",P);
if (edit_check_info ("Total:", "value", P*T) != E_OK)
    report_msg("The total is incorrect");
```

For more information, refer to the *TSL Online Reference*.

Default Checks and Specifying which Properties to Check

When you create a GUI checkpoint, you can determine the types of checks to perform on GUI objects in your application. For each object class, WinRunner recommends a default check. For example, if you select a push button, the default check determines whether the push button is enabled. Alternatively, in a dialog box you can specify which properties of an object to check. For example, you can choose to check a push button’s width, height, label, and position in a window (x- and y-coordinates).

To use the *default check*, you choose a Check GUI command on the Create menu. Click a window or an object in your application. WinRunner automatically captures information about the window or object and inserts a GUI checkpoint into the test script.

To specify which objects to check for an object, you choose a Check GUI command on the Create menu. Double-click a window or an object. In the Check GUI dialog box, choose the properties you want WinRunner to check. Click OK to save the checks and close the dialog box. WinRunner captures information about the GUI object and inserts a GUI checkpoint into the test script.

The following sections show the different object classes and the types of checks available.

Check Box Class

For a check box you can check the following properties:

X and Y: Checks the x-and y-coordinates of the top left corner of the button, relative to the window origin.

Width and Height: Checks the button’s width and height, in pixels.

Enabled: Checks whether the button can be selected.

Focused: Checks whether keyboard input will be directed to this button.

Label: Checks the button’s label.

State: Checks the button’s state (on or off) (default check).

Edit Class

For an edit object you can check the following properties:

X and Y: Checks the x-and y-coordinates of the top left corner of the edit object, relative to the window origin.

Width and Height: Checks the edit object's width and height, in pixels.

Enabled: Checks whether the edit object can be selected.

Focused: Checks whether keyboard input will be directed to this edit object.

Compare ("NUMBER"): Checks the contents of the edit field as a number.

The following checks require arguments. To set the arguments for a check, click the Add button next to it. The Property's Arguments dialog box opens and enables you to enter the arguments for the check.

Compare ("TEXT"): Checks the contents of the edit field as a text. Setting the Ignore Case argument to ON conducts a non-case-sensitive check on the contents of an edit field. The default value of the Ignore Case argument is OFF, which means that by default, checks on Edit Class objects are case sensitive.

Format ("DATE"): Checks that the contents of the edit field are in the specified date format. To specify a date format, select it from the dropdown list in the Property's Arguments dialog box.

Date Formats

WinRunner supports a wide range of date formats. These are given below, with an example for each.

Day dd Month yyyy	Thursday (or Thurs) 24 March (or Mar) 1998
Day, Month dd, yyyy	Thursday (or Thurs), March (or Mar) 24, 1998
dd Month yyyy	24 March 1998
dd-mm-yy	24-03-98
dd.mm.yy	24.03.98
dd/mm/yy	24/03/98
dd/mm/yyyy	24/03/1998
mm/dd/yy	03/24/98
yy/dd/mm	98/24/03
yyyy-mm-dd	1998-03-24

When the day or month begins with a zero (such as 03 for March), the 0 is not required for a successful format check.

Format (“TIME”): Checks that the contents of the edit field are in the specified time format. To specify the time format, select it from the dropdown list in the Property’s Arguments dialog box.

Time Formats

hh.mm.ss	10.20.56
hh:mm:ss	10:20:56
hh:mm:ss ZZ	10:20:56 A.M.

Range: Checks that the contents of the edit field are within the specified range. In the Property's Arguments dialog box, specify the lower limit in the top edit field, and the upper limit in the bottom edit field.

Regular Expression: Checks that the string in the edit field meets the requirements of the regular expression. In the Property's Arguments dialog box, enter a string into the Regular Expression box. Note that you do not need to precede the regular expression with an exclamation point. For more information, see Chapter 23, **Using Regular Expressions**.

List Class

For a list object you can check the following properties:

X and Y: Check the x-and y-coordinates of the top left corner of the list, relative to the window origin.

Width and Height: Checks the list’s width and height, in pixels.

Enabled: Checks whether an entry in the list can be selected.

Focused: Checks whether keyboard input will be directed to this list.

Selection: Checks the current list selection (default check).

Content: Checks the contents of the entire list.

Items_count: Checks the number of items in the list.

Menu Item Class

Menus cannot be accessed directly, by clicking them. To include a menu in a GUI checkpoint, click the window title bar. WinRunner prompts you to include all the objects in the window. Click Yes. The Add All dialog box opens. Select the Menus option. All menus in the window are added to the checklist, each menu item listed separately according to their default checks.

To make changes to a menu check, modify the GUI checklist after saving it. For more information, see **Editing Checklists** on page 157.

Has_sub_menu: Checks whether a menu item has a submenu.

Item_enabled: Checks whether the menu is enabled (default check).

Item_position: Checks the position of each item in the menu.

Sub_menus_count: Counts the number of items in the submenu.

Object Class

For an object that is not mapped to a standard object class, you can check the following properties:

X and Y: Checks the x-and y-coordinates of the top left corner of the GUI object, relative to the window origin (default checks).

Width and Height: Checks the object's width and height, in pixels (default checks).

Enabled: Checks whether the object can be selected.

Focused: Checks whether keyboard input will be directed to this object.

Push Button Class

For a push button you can check the following properties:

X and Y: Checks the x-and y-coordinates of the top left corner of the button, relative to the window origin.

Width and Height: Checks the button's width and height, in pixels.

Enabled: Checks whether the button can be selected (default check).

Focused: Checks whether keyboard input will be directed to this button.

Label: Checks the button's label.

Radio Button Class

The radio button properties are identical to the check button properties. see **Check Box Class** on page 184.

Scroll Class

For a scroll object you can check the following properties:

X and Y: Checks the x-and y-coordinates of the top left corner of the scroll, relative to the window origin.

Width and Height: Checks the scroll's width and height, in pixels.

Enabled: Checks whether the scroll can be selected.

Focused: Checks whether keyboard input will be directed to this scroll.

Position: Checks the current position of the scroll thumb within the scroll (default check).

Static Text Class

The static text properties are identical to the edit properties. See **Edit Class** on page 185.

Window Class

For a window you can check the following properties:

X and Y: Checks the x-and y-coordinates of the top left corner of the window.

Width and Height: Checks the window’s width and height, in pixels.

Enabled: Checks whether the window can be selected.

Focused: Checks whether keyboard input will be directed to this window.

Resizable: Checks whether the window can be resized.

Minimizable or Maximizable: Check whether the window can be minimized or maximized.

System_menu: Checks whether the window has a system menu.

Minimized or Maximized: Check whether the window is minimized or maximized.

Count objects: Counts the number of GUI objects in the window (default check).

Label: Checks the window’s label.

Find

?

Checking Property Values

You can create a GUI checkpoint that checks the value of a specific property of a GUI object. For example, you can check whether a button is enabled or disabled or whether an item in a list is selected. To create a GUI checkpoint for a property value, add one of the following functions to the test script:

button_check_info	**list_check_info**	**scroll_check_pos**
button_check_state	**list_check_item**	**static_check_info**
edit_check_info	**list_check_selected**	**static_check_text**
edit_check_selection	**obj_check_info**	**win_check_info**
edit_check_text	**scroll_check_info**	

For information about working with these functions, refer to the *TSL Online Reference*.

For example, **button_check_state** has the following syntax:

button_check_state (*button*, *state* **);**

The *button* is the logical name of the button. The *state* is the state of the button.

Suppose while testing the Flight Reservation application you order a plane ticket by typing in passenger and flight information. When all the relevant information has been entered, you click the Insert Order button to process the information. When you test this application, you want to check that once all the relevant information has been entered, the Insert Order button is enabled.

You can insert a **button_check_state** statement into your test script to check that the button is enabled before you click the Insert Order button. This function checks that the Insert Order button is enabled. You create this check to ensure that when you click the button, you know whether it is enabled. The following is a segment of the test script:

```
button_check_state("Insert Order",1);
button_press ("Insert Order");
```

Creating Tests
Working with ActiveX Controls

WinRunner supports Context Sensitive testing on ActiveX controls (also called OLE or OCX controls) in your application.

This chapter describes:

Find

?

About Working with ActiveX Controls

Many applications include ActiveX controls developed by third-party organizations. WinRunner can record and replay Context Sensitive operations on these controls, as well as check their properties. You can treat an ActiveX control as a generic object, as an object of a standard MSW class, or as an object of a new class that you define:

- By default, when WinRunner learns a description of an ActiveX control, it assigns it to the generic *object* class and uses the control of the property value for its logical name. User operations are recorded in the test script using the **obj_mouse_** TSL functions. By default, a GUI checkpoint checks the x- and y-coordinates and the width and height of the ActiveX control. When you create a GUI checkpoint, the logical name of the ActiveX control appears in the Objects column of the GUI Checkpoint dialog boxes. For more information, see Chapter 9, **Checking GUI Objects**.
- You can map an ActiveX control to a standard class of comparable behavior in order to enable WinRunner to treat your custom control as a standard GUI object. For more information on mapping a custom object to a standard class, see Chapter 6, **Configuring the GUI Map**.
- You can map an ActiveX control to a new class in order to check its unique properties. WinRunner enables you to check the properties that are unique to the ActiveX controls you use by using the **ole_** TSL functions. This enables you to greatly expand the range of information you can obtain from the ActiveX controls

in your application. Once property checks are enabled for a control, you can create a GUI checkpoint that compares the values of ActiveX control properties. Checking the properties of ActiveX controls is discussed in detail in this chapter.

To enable property checks for an ActiveX class, you can use the **ole_add_class_to_gui_ver** TSL function. WinRunner automatically inserts each **ole_add_class_to_gui_ver** statement that you execute into a script called *oleclass.* This script is called by the WinRunner startup test every time you start WinRunner.

At any time, you can view the current values of the properties of an ActiveX control using the OCX Properties Viewer. In addition, you can retrieve and define the values of properties for ActiveX controls using TSL functions.

Note: WinRunner provides special built-in support for checking the contents or properties of a table. If the ActiveX control you are checking is a table, see Chapter 11, **Checking Tables**.

Enabling Property Checks for an ActiveX Control Class

You use the TSL function **ole_add_class_to_gui_ver** to enable property checks for a class of ActiveX controls.

To enable property checks for an ActiveX control:

1 Choose Insert Function > From List on the Create menu to open the Function Generator dialog box. For information on using the Function Generator, see Chapter 17, **Using Visual Programming**.

2 Select ActiveX Objects from the Category list.

3 Select **ole_add_class_to_gui_ver** from the Function Name list.

4 Click Args to expand the dialog box.

Click the pointing hand button. The mouse pointer turns into a pointing hand.

5 Click an ActiveX control. Its logical name appears in the Name field.

6 In the edit field for the second argument, type in the names of properties to use as the default properties check.

7 Click Execute to execute **ole_add_class_to_gui_ver**.

WinRunner inserts the **ole_add_class_to_gui_ver** statement into the oleclass startup script. This script is called automatically by the WinRunner startup test every time you start WinRunner. For information on startup test scripts, see Chapter 36, **Initializing Special Configurations**.

Find
?

8 To add property checks for another ActiveX control class, repeat steps 5 through 7.

9 Click Close to close the Function Generator.

Checking ActiveX Control Properties with Default Checks

You can create a GUI checkpoint that checks the properties of an ActiveX control according to a list of properties that you defined as the default check. You define the default list of properties using the **ole_add_class_to_gui_ver** TSL function. You can also define the default list by selecting Save as Defaults in the Properties Verification dialog box (see **Checking ActiveX Control Properties with Selected Checks** on page 200).

To check ActiveX control properties with default checks:

1 Choose Check GUI > Object/Window on the Create menu. The WinRunner window is minimized to an icon, the mouse pointer turns into a pointing hand, and a help window opens.

2 Click the control once.

WinRunner captures the GUI information and stores it in the test's expected results directory. The WinRunner window is restored and an **obj_check_gui** statement is inserted in the test script.

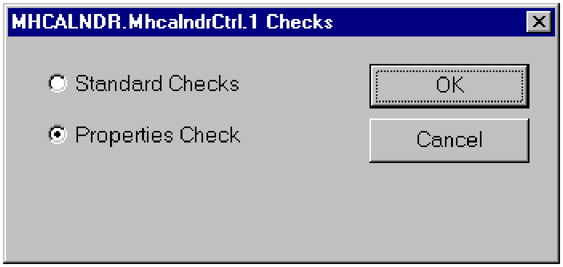
Checking ActiveX Control Properties with Selected Checks

You can create a GUI checkpoint that checks the properties of an ActiveX control according to properties that you select.

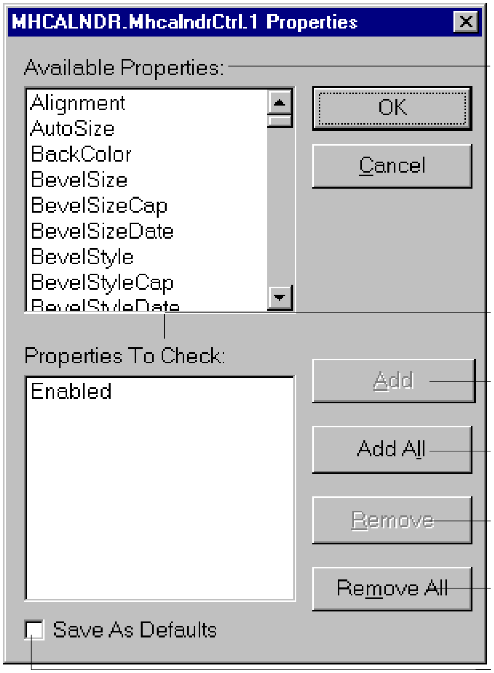
To check ActiveX control properties with selected checks:

1 Choose Check GUI > Object/Window on the Create menu. The WinRunner window is minimized to an icon, the mouse pointer turns into a pointing hand, and a help window opens.

2 Double-click the control. The Checks dialog box opens.



3 Make sure that Properties Check option is selected. Click OK. The Properties dialog box opens.



Lists all available properties for current check.

Lists selected properties for current check.

Adds selected properties to the list of properties to check.

Adds all properties to the list of properties to check.

Removes selected properties from the list of properties to check.

Removes all properties from the list of properties to check.

Saves selected properties as default properties.

4 Choose the properties to check:

Click one or more properties in the Available Properties list and click Add to move your selection to the Properties to Check list. Alternatively, double-click a single property to move it.

To check all available properties for the type of check you selected, click Add All.

5 Click OK to close the dialog box.

WinRunner captures the values of the selected properties and stores the information in the test's expected results directory. The WinRunner window is restored and an **obj_check_gui** statement is inserted into the test script.

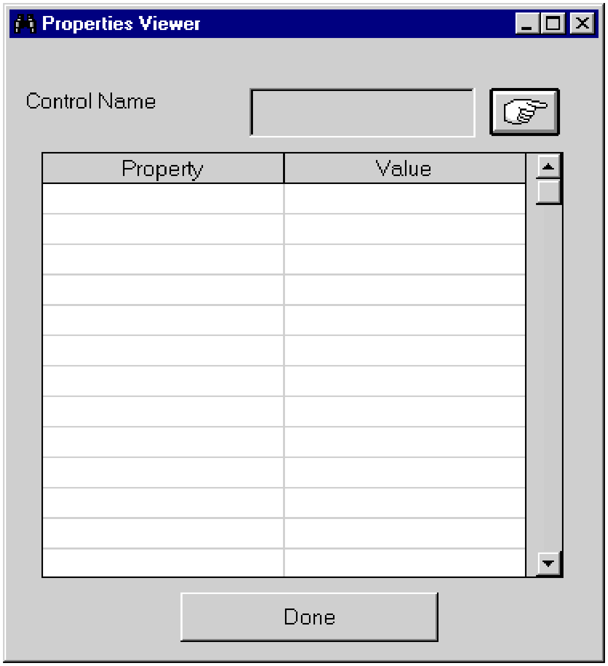
For more information on **obj_check_gui**, refer to the *TSL Online Reference*.

Viewing Properties

You use the OCX Properties Viewer to see the properties and property values for any ActiveX control. The OCX Properties Viewer is in the WinRunner program group.

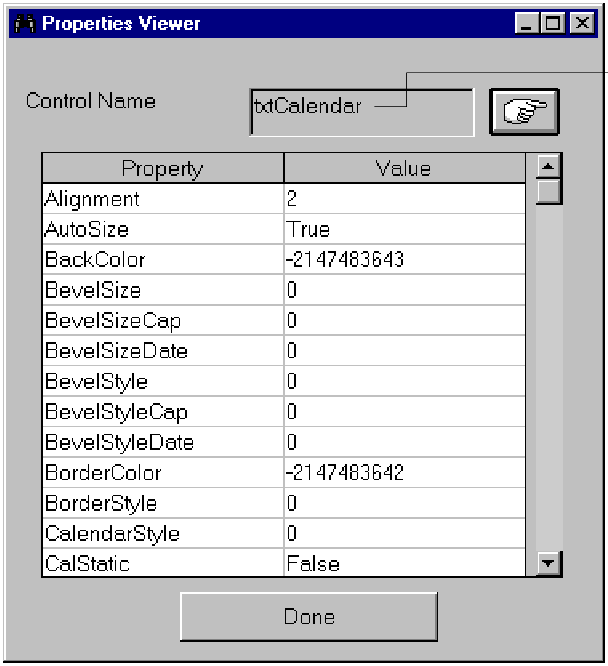
To view the properties of an ActiveX control:

1 Double-click the OCX Properties Viewer in the WinRunner program group to open the OCX Properties Viewer.



2 Click the pointing hand and click an ActiveX control.

3 The names and current values of the properties appear in the viewer.



Control Name: txtCalendar

Property	Value
Alignment	2
AutoSize	True
BackColor	-2147483643
BevelSize	0
BevelSizeCap	0
BevelSizeDate	0
BevelStyle	0
BevelStyleCap	0
BevelStyleDate	0
BorderColor	-2147483642
BorderStyle	0
CalendarStyle	0
CalStatic	False

In this example, this is a Visual Basic object in a Visual Basic application.

For other OCX controls, the class name appears in this box instead.

4 Click Done to close the dialog box.

Getting and Setting the Values of Properties

Two TSL functions enable you to retrieve and set the values of properties for ActiveX controls in your application. You can insert these functions into your test script using the Function Generator. For information on using the Function Generator, see Chapter 17, **Using Visual Programming**.

Getting the Value of an ActiveX Property

Use the **ole_obj_get_info** function to retrieve the value of any ActiveX control property. The syntax of this function is:

ole_obj_get_info (*object*, *property*, *out_value* **);**

Setting the Value of an ActiveX Property

Use the **ole_obj_set_info** function to set the value for any ActiveX control property. The syntax of this function is:

ole_obj_set_info (*object*, *property*, *out_value*, *type* **);**

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

Creating Tests
Checking Tables

When you work with WinRunner with added support for application development environments such as Visual Basic and PowerBuilder, you can create GUI checkpoints to check tables in your application.

This chapter describes:

About Checking Tables

Application development tools such as Visual Basic and PowerBuilder enable the creation of versatile client-server applications. These applications can display database information as a table. You can choose to install support for Visual Basic or PowerBuilder applications when you install WinRunner. The information in this chapter is also relevant if you have installed add-in support for other development environments, such as Oracle.

Once you install WinRunner support for any of these tools, you can add a GUI checkpoint to your test script that checks:

- *Contents* of a table
- ActiveX *Properties* of a table

If you are testing a PowerBuilder application, you can also check:

- Properties of a table's column
- Properties of a table's text
- Properties of table reports
- Contents of computed columns in the table
- Properties of computed columns in the table

Find

?

Note: In PowerBuilder applications, tables are called DataWindows. You can also check DropDown objects.

When you record operations on a table, WinRunner generates **tbl_** TSL statements in your test script. For example, if you click a cell, WinRunner generates a **tbl_set_selected_cell** statement. For more information on **tbl_** functions, refer to the *TSL Online Reference*.

You can create a GUI checkpoint for a table by pointing to it and choosing the properties that you want WinRunner to check. You can check the default properties recommended by WinRunner, or you can specify which properties to check. Information about the table and the properties to be checked is saved in a *checklist*. WinRunner then captures the current values of the table properties and saves this information as *expected results*. A GUI checkpoint is automatically inserted into the test script. This checkpoint appears in your test script as an **obj_check_gui** statement. For more information about GUI checkpoints and checklists, see Chapter 9, **Checking GUI Objects**.

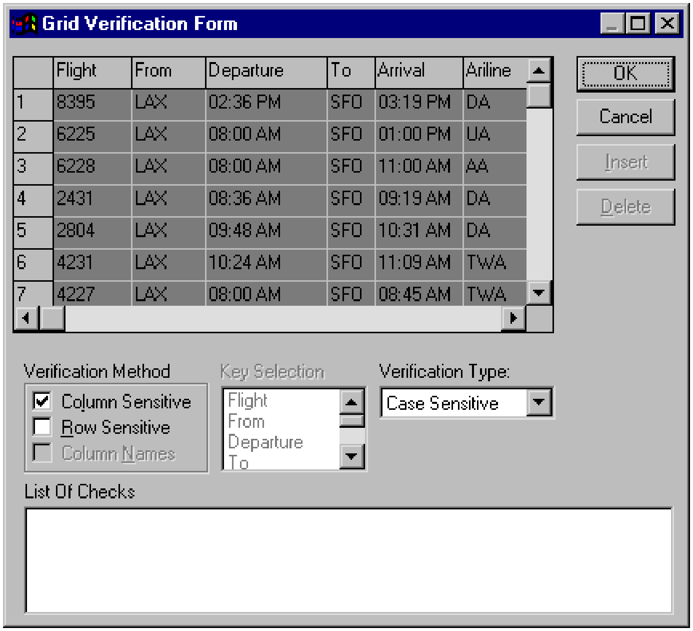
When you run the test, WinRunner compares the current state of the properties in the table to the expected results. If the expected results and the current results do not match, the GUI checkpoint fails. The results of the checkpoint can be viewed in the Test Results window. For more information, see Chapter 26, **Analyzing Test Results**.

Note that any GUI object you check that is not already in the GUI map is added automatically to the temporary GUI map file. See Chapter 3, **Introducing Context Sensitive Testing**, for more information.

This chapter provides step-by-step instructions for checking contents of tables, contents of PowerBuilder dropdown objects, and table properties. It also shows how you can use TSL functions to define and retrieve information in tables in your application.

Table Contents Verification

The Grid Verification form opens when you double-click in a table while creating a GUI checkpoint in order to select which checks to perform on the table. The Grid Verification form displays the content of your table as a grid. You can use several verification types and methods to check the contents.



	Flight	From	Departure	To	Arrival	Ariline
1	8395	LAX	02:36 PM	SFO	03:19 PM	DA
2	6225	LAX	08:00 AM	SFO	01:00 PM	UA
3	6228	LAX	08:00 AM	SFO	11:00 AM	AA
4	2431	LAX	08:36 AM	SFO	09:19 AM	DA
5	2804	LAX	09:48 AM	SFO	10:31 AM	DA
6	4231	LAX	10:24 AM	SFO	11:09 AM	TWA
7	4227	LAX	08:00 AM	SFO	08:45 AM	TWA

Verification Method
- [✓] Column Sensitive
- [ ] Row Sensitive
- [ ] Column Names

Key Selection: Flight, From, Departure, To

Verification Type: Case Sensitive

List Of Checks

Verification Type

WinRunner can verify the contents of a table in several different ways.

- **Case Sensitive** (the default): WinRunner compares the text content of the selection, ignoring any differences in font size, type, and color. Any difference in case or text content between the expected and actual data results in a mismatch.
- **Case Insensitive:** WinRunner compares the text content of the selection, ignoring any differences in font size, type, color, and case. Only differences in text content between the expected and actual data result in a mismatch.
- **Numeric Content:** WinRunner evaluates the selected data according to numeric values. WinRunner recognizes, for example, that “2” and “2.00” are the same number. It also ignores the alignment of numerals within the cell.
- **Numeric Range:** WinRunner compares the selected data against a numeric range. Both the minimum and maximum values are any real number that you specify. This comparison differs from text and numeric content verification in that the actual table data is compared against the range that you defined and not against the expected results.

Verification Method

Several different verification methods are available. These let you control how WinRunner identifies columns or rows within a table.

For a Multiple-Column Table

- **Column Sensitive** (the default setting): WinRunner looks for the selection according to the column names. A shift in the position of the columns within the table does not result in a mismatch.
- **Row Sensitive:** WinRunner looks for the rows in the selection according to a unique identifier, called a key. The **Key Selection** lists the name of all columns in the table. A shift in the position of any of the rows does not result in a mismatch.
- **Column Names:** WinRunner checks the selected cells according to their physical position in the table. The column names are included in the verification. To enable this option, clear the Column Sensitive option.

For a Single-Column Table

- **By Position:** WinRunner checks the selection according to the location of the items within the list.
- **By Content:** WinRunner checks the selection according to the content of the items, ignoring their location in the list.

Checking Table Contents with Default Checks

The default GUI check on a table is a case-sensitive and column-sensitive check of the table’s text.

To check the contents of a table with default checks:

1 Choose Check GUI > Object/Window on the Create menu.

The WinRunner window is minimized to an icon, the mouse pointer turns into a pointing hand, and a help window opens.

2 Click once in the table.

WinRunner captures the GUI information and stores it in the test’s expected results directory. The WinRunner window is restored and an **obj_check_gui** statement is inserted into the test script.

Checking Table Contents with Selected Checks

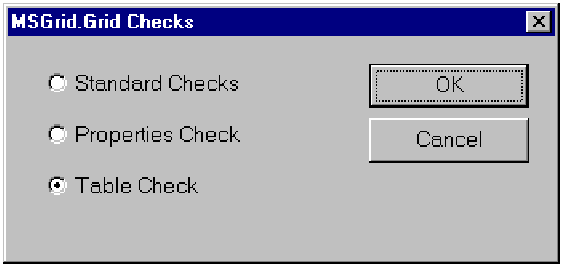
You can create a GUI checkpoint that performs selected checks on the contents of a table.

To check the contents of a table with selected checks:

1 Choose Check GUI > Object/Window on the Create menu.

The WinRunner window is minimized to an icon, the mouse pointer becomes a pointing hand, and a help window opens.

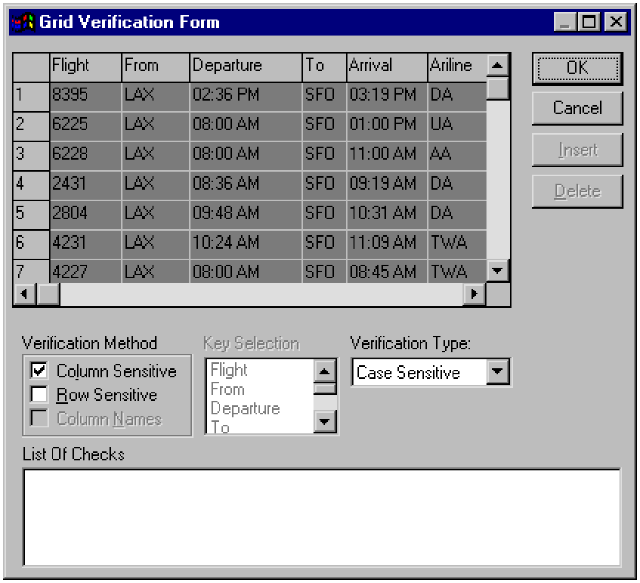
Double-click in the table. For Visual Basic applications or applications with OCX support, the MSGrid Grid Check dialog box opens, as is shown below. For PowerBuilder applications, a similar dialog box opens.


MSGrid.Grid Checks

○ Standard Checks
○ Properties Check
● Table Check

Find
?

2 Make sure the Table Check option is selected in the dialog box. Click OK. The Grid Verification form opens.


Grid Verification Form

	Flight	From	Departure	To	Arrival	Ariline
1	8395	LAX	02:36 PM	SFO	03:19 PM	DA
2	6225	LAX	08:00 AM	SFO	01:00 PM	UA
3	6228	LAX	08:00 AM	SFO	11:00 AM	AA
4	2431	LAX	08:36 AM	SFO	09:19 AM	DA
5	2804	LAX	09:48 AM	SFO	10:31 AM	DA
6	4231	LAX	10:24 AM	SFO	11:09 AM	TWA
7	4227	LAX	08:00 AM	SFO	08:45 AM	TWA

Verification Method: ☑ Column Sensitive ☐ Row Sensitive ☐ Column Names

Key Selection: Flight, From, Departure, To

Verification Type: Case Sensitive

List Of Checks

The Grid Verification form displays the table as a grid. The rows are numbered, and the column names are taken from the database.

3 Select a verification type from the Verification Type list. If you choose Row Sensitive, select one or more keys from the Key Selection list.

4 Specify the parts of the table to include in the verification:

To insert *the entire table*, double-click the upper left cell or click it once and then click Insert.

To insert *one or more full columns*, double-click the column header(s) or click once and then click Insert.

To insert *one or more full rows*, double-click the row number or click it once and then click Insert.

To insert *any block of cells*, highlight the block and then click Insert.

A description of the checks inserted appears in the List of Checks field at the bottom of the dialog box.

5 To add more checks, repeat steps 3 and 4. (You may choose a different verification *type* for each check.) To remove a check from the list, double-click the description of the check or select it and press Delete.

6 Select a verification *method* for the entire list of checks and click OK to close the dialog box.

Note: The verification *method* applies to the entire list of checks. You cannot select different methods for different checks in a list.

WinRunner captures the GUI information and stores it in the test's expected results directory. The WinRunner window is restored and an **obj_check_gui** statement is inserted into the test script.

Checking Contents of a Drop-Down Object

For a PowerBuilder application, you can create a GUI checkpoint that checks the contents of a drop-down object. A drop-down object displays database contents as a list.

To check the contents of a drop-down object:

1. Choose Check GUI > Object/Window on the Create menu. The WinRunner window is minimized to an icon, the mouse pointer turns into a pointing hand, and a help window opens.
2. To check the drop-down object with its default checks, click on it once.

   To define a custom check, double-click the drop-down object. The Check GUI dialog box opens.
3. Select the type of checks.
4. To save the checklist and close the dialog box, click OK.

WinRunner captures the GUI information and stores it in the test's expected results directory. The WinRunner window is restored and an **obj_check_gui** statement is inserted into the test script. For more information on the **obj_check_gui** function, refer to the *TSL Online Reference*.

Checking Table Properties

The appearance and behavior of tables in a database application are determined by the *properties* defined by the developer. WinRunner lets you see the values of these properties and check them using a GUI checkpoint in your test script.

To create a GUI checkpoint of table properties, you point to a table, indicate what kind of property check to create, and select the properties to check. WinRunner captures the current values of the properties and saves this information as *expected results*. A GUI checkpoint is automatically inserted into the test script as an **obj_check_gui** statement.

When you run the test, WinRunner compares the current values of the properties to the expected results. If the expected and current results do not match, the GUI checkpoint fails. For more information, see Chapter 26, **Analyzing Test Results**.

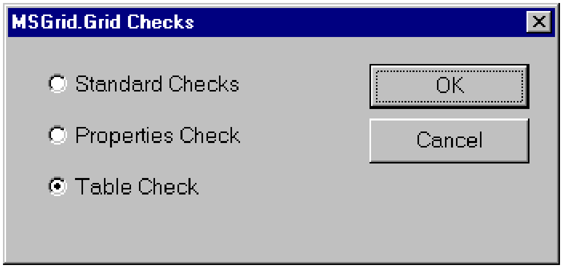
To create a GUI checkpoint for table properties:

1 Choose Check GUI > Object/Window on the Create menu.

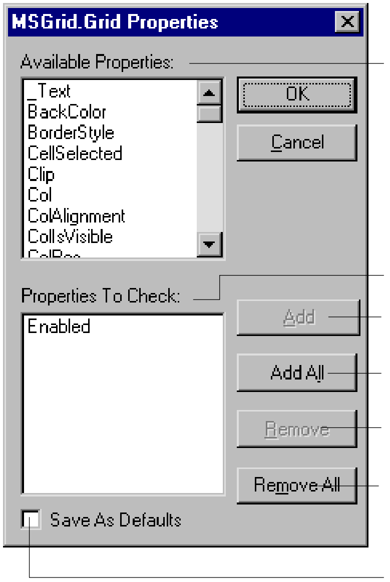
The WinRunner window is minimized to an icon, the mouse pointer turns into a pointing hand, and a help window opens.

Find
?

2 Double-click the table. For Visual Basic applications or applications with OCX support, the MSGrid Grid Checks dialog box opens, as is shown below. For PowerBuilder applications, a similar dialog box opens.



3 Select Properties Check and click OK. The MSGrid Properties dialog box opens.



Lists all available properties for current check.

Lists selected properties for current check.

Adds selected properties to list of properties to check.

Adds all properties to list of properties to check.

Removes selected properties from list of properties to check.

Removes all properties from list of properties to check.

Saves selected properties as default properties.

4 Choose the properties to check for the table:

Click one or more properties in the Available Properties list and click Add to move your selection to the Properties to Check list. Alternatively, double-click a single property to move it.

To check all available properties, click Add All.

5 Click OK to close the dialog box.

WinRunner captures the values of the selected properties and stores the information in the test's expected results directory. The WinRunner window is restored and an **obj_check_gui** statement is inserted into the test script.

For more information on **obj_check_gui**, refer to the *TSL Online Reference*.

Using TSL Functions with Tables and Calendars

WinRunner provides several TSL functions that enable you to obtain and define information in tables and calendars in your application.

You can use the Function Generator to insert the following functions into your test scripts, or you can manually program statements that use these functions.

Note: The availability and exact syntax definitions of these functions may vary among toolkits. Refer to the *TSL Online Reference* for further details and for examples of how to use the functions in a test script.

Getting Cell Contents

The **tbl_get_cell_data** function retrieves the contents of the specified cell from a table. This function has the following syntax:

tbl_get_cell_data (*table*, *row*, *column*, *out_text* **);**

The *table* is the logical name of the table. The *row* specifies the location of the row within the table. The *column* specifies either the location of the column within the table or the name of the column. The *out_text* is the output variable that stores the string found in the specified cell. For PowerBuilder users, the *row* can also specify the contents of one or more cells in the row in the table.

The **tbl_get_cell_data** function is supported for the following OCXs:

OCX	**MSW_class**
Data Bound Grid Control	MSDBGrid.DBGrid
FarPoint Spreadsheet Control	FPSpread.Spread.1
MicroHelp MH3d List Control	MHGLBX.Mh3dListCtrl.1
Microsoft Grid Control	MSGrid.Grid
Sheridan Data Grid Control	SSDataWidgets.SSDBGridCtrl.1
True DBGrid Control	TrueDBGrid50.TDBGrid

Setting Cell Contents

The **tbl_set_cell_data** function sets the contents of the specified cell in a table. This function has the following syntax:

tbl_set_cell_data (*table*, *row*, *column*, *text* **);**

The *table* is the logical name of the table. The *row* specifies the location of the row within the table. The *column* specifies either the location of the column within the table or the name of the column. The *text* is the variable that stores the string set in the specified cell. For PowerBuilder users, the *row* can also specify the contents of one or more cells in the row in the table.

The **tbl_set_cell_data** function is supported for the following OCXs:

OCX	**MSW_class**
Data Bound Grid Control	MSDBGrid.DBGrid
FarPoint Spreadsheet Control	FPSpread.Spread.1
MicroHelp MH3d List Control	MHGLBX.Mh3dListCtrl.1
Microsoft Grid Control	MSGrid.Grid
Sheridan Data Grid Control	SSDataWidgets.SSDBGridCtrl.1
True DBGrid Control	TrueDBGrid50.TDBGrid

Getting the Cell Currently in Focus

The **tbl_get_selected_cell** function retrieves the cell that is currently in focus in a table. This function has the following syntax:

tbl_get_selected_cell (*table*, *out_row*, *out_column* **);**

The *table* is the logical name of the table. The *out_row* is the output variable that stores the location of the row within the table. The *out_column* is the output variable that stores the column name of the specified cell. For PowerBuilder users, the out_*row* can also specify the contents of one or more cells in the row in the table.

The **tbl_get_selected_cell** function is supported for the following OCXs:

OCX	**MSW_class**
Data Bound Grid Control	MSDBGrid.DBGrid
FarPoint Spreadsheet Control	FPSpread.Spread.1
MicroHelp MH3d List Control	MHGLBX.Mh3dListCtrl.1
Microsoft Grid Control	MSGrid.Grid
Sheridan Data Grid Control	SSDataWidgets.SSDBGridCtrl.1
True DBGrid Control	TrueDBGrid50.TDBGrid

Find
?

Selecting a Cell

The **tbl_set_selected_cell** function sets the specified cell in a table as the current cell. This is equivalent to clicking a cell to bring it into focus. This function has the following syntax:

tbl_set_selected_cell (*table*, *row*, *column* **);**

The *table* is the logical name of the table. The *row* specifies the location of the row within the table. The *column* specifies either the location of the column within the table or the name of the column. For PowerBuilder users, the *row* can also specify the contents of one or more cells in the row in the table.

The **tbl_set_selected_cell** function is supported for the following OCXs:

OCX	**MSW_class**
Data Bound Grid Control	MSDBGrid.DBGrid
FarPoint Spreadsheet Control	FPSpread.Spread.1
MicroHelp MH3d List Control	MHGLBX.Mh3dListCtrl.1
Microsoft Grid Control	MSGrid.Grid
Sheridan Data Grid Control	SSDataWidgets.SSDBGridCtrl.1
True DBGrid Control	TrueDBGrid50.TDBGrid

Activating a Cell

The **tbl_activate_cell** function sets the specified cell in a table as the activated cell. It is equivalent to double-clicking a cell to activate it. This function has the following syntax:

tbl_activate_cell (*table*, *row*, *column* **);**

The *table* is the logical name of the table. The *row* specifies the location of the row within the table. The *column* specifies either the location of the column within the table or the name of the column. For PowerBuilder users, the *row* can also specify the contents of one or more cells in the row in the table.

The **tbl_activate_cell** function is supported for the following OCXs:

OCX	MSW_class
Data Bound Grid Control	MSDBGrid.DBGrid
FarPoint Spreadsheet Control	FPSpread.Spread.1
MicroHelp MH3d List Control	MHGLBX.Mh3dListCtrl.1
Microsoft Grid Control	MSGrid.Grid
Sheridan Data Grid Control	SSDataWidgets.SSDBGridCtrl.1
True DBGrid Control	TrueDBGrid50.TDBGrid

Getting the Number of Columns

The **tbl_get_cols_count** function retrieves the number of columns in a table. This function has the following syntax:

tbl_get_cols_count (*table*, *out_cols_count* **);**

The *table* is the logical name of the table. The *out_cols_count* is the output variable that stores the total number of columns in the table.

The **tbl_get_cols_count** function is supported for the following OCXs:

OCX	MSW_class
Data Bound Grid Control	MSDBGrid.DBGrid
FarPoint Spreadsheet Control	FPSpread.Spread.1
MicroHelp MH3d List Control	MHGLBX.Mh3dListCtrl.1
Microsoft Grid Control	MSGrid.Grid
Sheridan Data Grid Control	SSDataWidgets.SSDBGridCtrl.1
True DBGrid Control	TrueDBGrid50.TDBGrid

Getting the Names of Columns

The **tbl_get_column_name** function retrieves the column header name of the specified column in a table. This function has the following syntax:

tbl_get_column_name (*table*, *col_index*, *out_col_names* **);**

The *table* is the logical name of the table. The *col_index* is the numeric index of the column in the table. The *out_col_names* is the output variable that stores the name of the specified column in the table.

The **tbl_get_column_name** function is supported for the following OCXs:

OCX	**MSW_class**
Data Bound Grid Control	MSDBGrid.DBGrid
FarPoint Spreadsheet Control	FPSpread.Spread.1
MicroHelp MH3d List Control	MHGLBX.Mh3dListCtrl.1
Microsoft Grid Control	MSGrid.Grid
Sheridan Data Grid Control	SSDataWidgets.SSDBGridCtrl.1
True DBGrid Control	TrueDBGrid50.TDBGrid

Find

?

Getting the Number and Names of Columns

For PowerBuilder DataWindows, the **tbl_get_column_names** function retrieves the names and number of columns in a table. This function has the following syntax:

tbl_get_column_names (*table*, *out_col_names*, *out_cols_count* **);**

The *table* is the logical name of the table. The *out_col_names* is the output variable that stores the names of the columns in the table. The *out_cols_count* is the output variable that stores the total number of columns in the table.

Selecting all the Cells in a Column

The **tbl_select_col_header** function selects all the cells in the specified column of a table. This is equivalent to clicking the column header to bring all the cells in the column into focus. This function has the following syntax:

tbl_select_col_header (*table*, *column* **);**

The *table* is the logical name of the table. The *column* specifies either the location of the column within the table or the name of the column.

The **tbl_select_col_header** function is supported for the following OCXs:

OCX	**MSW_class**
Data Bound Grid Control	MSDBGrid.DBGrid
FarPoint Spreadsheet Control	FPSpread.Spread.1
MicroHelp MH3d List Control	MHGLBX.Mh3dListCtrl.1
Microsoft Grid Control	MSGrid.Grid
Sheridan Data Grid Control	SSDataWidgets.SSDBGridCtrl.1
True DBGrid Control	TrueDBGrid50.TDBGrid

Activating all the Cells in a Column

The **tbl_activate_header** function activates all the cells in the specified column of a table. This is equivalent to double-clicking the column header to activate all the cells in the column. This function has the following syntax:

tbl_activate_header (*table*, *column* **);**

The *table* is the logical name of the table. The *column* specifies either the location of the column within the table or the name of the column.

The **tbl_activate_header** function is supported for the following OCXs:

OCX	MSW_class
Data Bound Grid Control	MSDBGrid.DBGrid
FarPoint Spreadsheet Control	FPSpread.Spread.1
MicroHelp MH3d List Control	MHGLBX.Mh3dListCtrl.1
Microsoft Grid Control	MSGrid.Grid
Sheridan Data Grid Control	SSDataWidgets.SSDBGridCtrl.1
True DBGrid Control	TrueDBGrid50.TDBGrid

Getting the Number of Rows

The **tbl_get_rows_count** function retrieves the number of rows in a table. This function has the following syntax:

tbl_get_rows_count (*table*, *out_rows_count* **);**

The *table* is the logical name of the table. The *out_rows_count* is the output variable that stores the total number of rows in the table.

The **tbl_get_rows_count** function is supported for the following OCXs:

OCX	**MSW_class**
FarPoint Spreadsheet Control	FPSpread.Spread.1
MicroHelp MH3d List Control	MHGLBX.Mh3dListCtrl.1
Microsoft Grid Control	MSGrid.Grid
Sheridan Data Grid Control	SSDataWidgets.SSDBGridCtrl.1
True DBGrid Control	TrueDBGrid50.TDBGrid

Getting the Row Currently in Focus

The **tbl_get_selected_row** function retrieves the row that is currently selected in a table. This function has the following syntax:

tbl_get_selected_row (*table*, *out_row* **);**

The *table* is the logical name of the table. The *out_row* is the output variable that stores either the location of the row within the table or the contents of one or more columns in the table.

The **tbl_get_selected_row** function is supported for the following OCXs:

OCX	MSW_class
Data Bound Grid Control	MSDBGrid.DBGrid
FarPoint Spreadsheet Control	FPSpread.Spread.1
MicroHelp MH3d List Control	MHGLBX.Mh3dListCtrl.1
Sheridan Data Grid Control	SSDataWidgets.SSDBGridCtrl.1
True DBGrid Control	TrueDBGrid50.TDBGrid

Selecting a Row

The **tbl_set_selected_row** function specifies the row to be in focus in a table. This is equivalent to clicking a row to bring it into focus. This function has the following syntax:

tbl_set_selected_row (*table*, *row* **);**

The *table* is the logical name of the table. The *row* specifies the location of the row within the table. For PowerBuilder users, the *row* can also specify the contents of one or more cells in the row in the table.

The **tbl_set_selected_row** function is supported for the following OCXs:

OCX	MSW_class
Data Bound Grid Control	MSDBGrid.DBGrid
FarPoint Spreadsheet Control	FPSpread.Spread.1
MicroHelp MH3d List Control	MHGLBX.Mh3dListCtrl.1
Microsoft Grid Control	MSGrid.Grid
True DBGrid Control	TrueDBGrid50.TDBGrid

Selecting a Date in a Calendar

The **calendar_select_date** function selects the specified date in a calendar. This is equivalent to clicking a date to bring it into focus. This function has the following syntax:

calendar_select_date (*calendar*, *date* **);**

The *calendar* is the logical name of the calendar. The *date* specifies the date.

Activating a Date in a Calendar

The **calendar_activate_date** function activates the specified date in a calendar. This is equivalent to double-clicking a date to activate it. This function has the following syntax:

calendar_activate_date (*calendar*, *date* **);**

The *calendar* is the logical name of the calendar. The *date* specifies the date.

Find
?

Getting and Counting Dates in a Calendar

The **calendar_get_selected** function retrieves and counts the dates selected in a calendar. This function has the following syntax:

calendar_get_selected (*calendar*, *out_dates*, *out_dates_count* **);**

The *calendar* is the logical name of the calendar. The *out_dates* is the output variable that stores the selected dates in the calendar. The *out_dates_count* is the output variable that stores the total number of selected dates in the calendar.

Find

?

Using TSL Table Functions with OCXs

The TSL Table functions are supported for a number of OCXs. The OCXs and the functions for which they are supported are listed in the table on the next page. For a description of each function, see **Using TSL Functions with Tables and Calendars** on page 222. For detailed information about each function and examples of usage, refer to the *TSL Online Reference*.

	Data Bound Grid Control	FarPoint Spreadsheet Control	MicroHelp MH3d List Control	Microsoft Grid Control	Sheridan Data Grid Control	True DBGrid Control
tbl_activate_cell	✔	✔	✔	✔	✔	✔
tbl_activate_header	✔	✔	✔	✔	✔	✔
tbl_get_cell_data	✔	✔	✔	✔	✔	✔
tbl_get_cols_count	✔	✔	✔	✔	✔	✔
tbl_get_column_name	✔	✔	✔	✔	✔	✔
tbl_get_rows_count		✔	✔	✔	✔	✔
tbl_get_selected_cell	✔	✔	✔	✔	✔	✔
tbl_get_selected_row	✔	✔	✔		✔	✔
tbl_select_col_header	✔	✔	✔	✔	✔	✔
tbl_set_cell_data	✔	✔	✔	✔	✔	✔
tbl_set_selected_cell	✔	✔	✔	✔	✔	✔
tbl_set_selected_row	✔	✔	✔	✔		✔

Checking Bitmaps

WinRunner enables you to compare two versions of an application being tested by matching captured bitmaps. This is particularly useful for checking non-GUI areas of your application, such as drawings or graphs.

This chapter describes:

- **Checking Window and Object Bitmaps**
- **Checking Area Bitmaps**

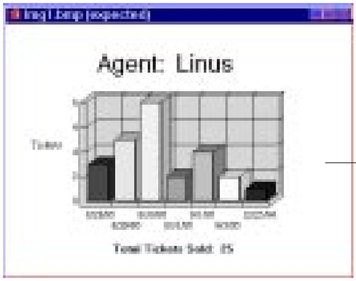
About Checking Bitmaps

You can check an object, a window, or an area of a screen in your application as a bitmap. While creating a test, you indicate what you want to check. WinRunner captures the specified bitmap, stores it in the expected results directory (*exp*) of the test, and inserts a checkpoint in the test script. When you run the test, WinRunner compares the bitmap currently displayed in the application being tested with the *expected* bitmap stored earlier. In the event of a mismatch, WinRunner captures the current *actual* bitmap and generates a *difference* bitmap. By comparing the three bitmaps (expected, actual, and difference), you can identify the nature of the discrepancy.

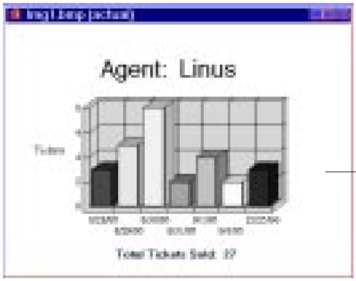
Suppose, for example, your application includes a graph that displays database statistics. You could capture a bitmap of the graph in order to compare it with a bitmap of the graph from a different version of your application. If there is a difference between the graph captured for expected results and the one captured during the test run, WinRunner generates a bitmap that shows the difference, pixel by pixel.

Find

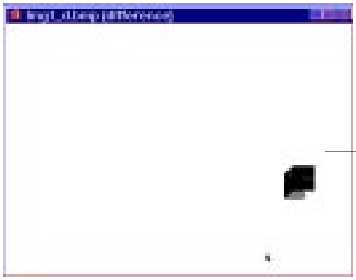
?



In the expected graph, captured when the test was created, 25 tickets were sold.



In the actual graph, captured during the test run, 27 tickets were sold. The last column is taller because of the larger quantity. of tickets.



The difference bitmap shows where the two graphs diverged: in the height of the last column, and in the number of tickets sold.

When working in Context Sensitive mode, you can capture a bitmap of a window, object, or of a specified area of a screen. WinRunner inserts a checkpoint in the test script in the form of either a **win_check_bitmap** or **obj_check_bitmap statement**.

To check a bitmap, you start by selecting Check Bitmap from the Create menu. To capture a window or another GUI object, you click it with the mouse. To capture an area bitmap, you mark the area to be checked using a crosshairs mouse pointer.

Note that when you record a test in Analog mode, you should press the CHECK BITMAP softkey or the CHECK BITMAP AREA softkey to create a bitmap checkpoint. This prevents WinRunner from recording extraneous mouse movements. If you are programming a test, you can also use the Analog function **check_window** to check a bitmap. For more information refer to the *TSL Online Reference*.

If the name of a window or object varies each time you run a test, you can define a regular expression in the GUI Map Editor. This instructs WinRunner to ignore all or part of the name. For more information on using regular expressions in the GUI Map Editor, see Chapter 5, **Editing the GUI Map**.

Note: You cannot use bitmap checkpoints created in XRunner when you run a test script in WinRunner. You must recreate these checkpoints in WinRunner. For information on using GUI maps created in XRunner in WinRunner test scripts, see Chapter 6, **Configuring the GUI Map**. For information on using XRunner test scripts recorded in Analog mode, see Chapter 8, **Creating Tests**. For information on using GUI checkpoints created in XRunner, see Chapter 9, **Checking GUI Objects**.

Find

?

Checking Window and Object Bitmaps

You can capture a bitmap of any window or object in your application by pointing to it. The method for capturing objects and for capturing windows is identical. You start by choosing Check Bitmap > Object/Window on the Create menu. As you pass the mouse pointer over the windows of your application, objects and windows flash. To capture a window bitmap, you click the window's title bar. To capture an object within a window as a bitmap, you click the object itself.

Note that during recording, when you capture an object in a window that is not the active window, WinRunner automatically generates a **set_window** statement.

To capture a window or object as a bitmap:

1 Choose Check Bitmap > Object/Window on the Create menu. Alternatively, if you are recording in Analog mode, press the CHECK BITMAP softkey.

  The WinRunner window is minimized to an icon, the mouse pointer becomes a pointing hand, and a help window opens.

2 Point to the object or window and click it. WinRunner captures the bitmap and generates a **win_check_bitmap** or **obj_check_bitmap** statement in the script.

The TSL statement generated for a window bitmap has the following syntax:

win_check_bitmap (*object, bitmap, time*);

For an object bitmap, the syntax is:

obj_check_bitmap (*object, bitmap, time*);

For example, when you click the title bar of the main window of the Flight Reservation application, the resulting statement might be:

win_check_bitmap ("Flight Reservation", "Img2", 1);

However, if you click the Date of Flight box in the same window, the statement might be:

obj_check_bitmap ("Date of Flight:", "Img1", 1);

For more information on the **win_check_bitmap** and **obj_check_bitmap** functions, refer to the *TSL Online Reference.*

Note: The execution of the **win_check_bitmap** and **obj_check_bitmap functions** is affected by the current values specified for the *delay*, *timeout* and *min_diff* testing options. For more information on these testing options and how to modify them, see Chapter 34, **Setting Testing Options from a Test Script**. You can also set the corresponding Delay for Window Synchronization, Timeout for Checkpoints and CS Statements, and Threshold for Difference between Bitmaps testing options globally using the Options dialog box. For more information, see Chapter 33, **Setting Global Testing Options**.

Find

?

Checking Area Bitmaps

You can define any rectangular area of the screen and capture it as a bitmap for comparison. The area can be any size; it can be part of a single window, or it can intersect several windows. The rectangle is identified by the coordinates of its upper left and lower right corners, relative to the upper left corner of the window in which the area is located. If the area intersects several windows or is part of a window with no title (for example, a popup window), its coordinates are relative to the entire screen (the root window).

Find

?

To capture an area of the screen as a bitmap:

1 Choose Check Bitmap > Area on the Create menu. Alternatively, if you are recording in Analog mode, press the CHECK BITMAP AREA softkey. The WinRunner window is minimized to an icon, the mouse pointer becomes a crosshairs pointer, and a help window opens.

2 Mark the area to be captured: click the left mouse button and drag the mouse pointer until a rectangle encloses the area, then release the mouse button.

3 Click the right mouse button to complete the operation. WinRunner captures the area and generates a **win_check_bitmap** statement in your script.

Note: Execution of the **win_check_bitmap** function is affected by the current settings specified for the *delay*, *timeout* and *min_diff* test options. For more information on these testing options and how to modify them, see Chapter 34, **Setting Testing Options from a Test Script**. You can also set the corresponding Delay for Window Synchronization, Timeout for Checkpoints and CS Statements, and Threshold for Difference between Bitmaps testing options globally using the Options dialog box. For more information, see Chapter 33, **Setting Global Testing Options**.

The **win_check_bitmap** statement for an area of the screen has the following syntax:

```
win_check_bitmap ( window, bitmap, time, x, y, width, height );
```

For example, when you define an area to check in the Flight Reservation application, the resulting statement might be:

```
win_check_bitmap ("Flight Reservation", "Img3", 1, 9, 159, 104, 88);
```

For more information on **win_check_bitmap**, refer to the *TSL Online Reference*.

Checking Text

WinRunner allows you to read and check text in a GUI object or in any area of your application.

This chapter describes:

Find

?

About Checking Text

You can use text checkpoints in your test scripts to read and check text in GUI objects and in areas of the screen. While creating a test you point to an object or window containing text. WinRunner reads the text and writes a TSL statement to the test script. You may then add simple programming elements to your test scripts to verify the contents of the text.

You can use a text checkpoint to:

- read text from a GUI object or window in your application, using **obj_get_text** and **win_get_text**
- search for text in an object or window, using **win_find_text** and **obj_find_text**
- move the mouse pointer to text in an object or window, using **obj_move_locator_text** and **win_move_locator_text**
- click on text in an object or window, using **obj_click_on_text** and **win_click_on_text.**
- compare two strings, using **compare_text**

Note that you should use a text checkpoint on a GUI object only when a GUI checkpoint cannot be used to check the text. For example, suppose you want to check the text on a custom graph object. Since this custom object cannot be mapped to a standard object class (such as pushbutton, list, or menu), WinRunner associates it with the general object class. A GUI checkpoint for such

Find
?

an object can check only the object’s width, height, x- and y- coordinates, and whether the object is enabled or focused. It cannot check the text in the object. To do so, you must create a text checkpoint.

The following script segment uses the **win_get_text** function to read text in a graph in a Flight Reservation application.

```
set_window ("Graph", 10);
win_get_text ("Graph", text);
if (text=="Total Tickets Sold: 26")
        report_msg ("The total is correct.");
```

WinRunner can read the visible text from the screen in nearly any situation. Usually this process is automatic. In certain situations, however, WinRunner must first learn the fonts used by your application. Use the Learn Fonts utility to teach WinRunner the fonts. An explanation of when and how to perform this procedure appears in **Teaching Fonts to WinRunner** on page 263.

Reading Text

You can read the entire text contents of any GUI object or window in your application, or the text in a specified area of the screen. You read text using the **win_get_text, obj_get_text, and get_text** functions. These functions can be generated automatically, using the Get Text command on the Create menu, or manually, by programming. In both cases, the read text is assigned to an output variable.

To read all the text in a GUI object, you choose Get Text > Object/Window and click an object with the mouse pointer. To read the text in an area of an object or window, you choose Get Text > Area and then use a crosshairs pointer to enclose the text in a rectangle.

In most cases, WinRunner can identify the text on GUI objects automatically. However, if you try to read text and the comment #no text was found is inserted into the test script, this means WinRunner was unable to identify your application font. To enable WinRunner to identify text, you must teach WinRunner your application fonts. For more information, see **Teaching Fonts to WinRunner** on page 263.

Reading All the Text in a Window or Object

You can read all the visible text in a window or other object using **win_get_text** or **obj_get_text**.

To read all the visible text in a window or object:

1 Choose Get Text > Object/Window on the Create menu. Alternatively, if you are recording in Analog mode, press the GET TEXT OBJECT softkey.

WinRunner is minimized to an icon, the mouse pointer becomes a pointing hand, and a Help window opens.

2 Click the window or object. WinRunner captures the text in the object and generates a **win_get_text or obj_get_text** statement.

In the case of a window, this statement has the following syntax:

win_get_text (*window*, *text* **);**

The *window* is the name of the window. The *text* is an output variable that holds all of the text displayed in the window. To make your script easier to read, this text is inserted into the script as a comment when the function is recorded.

For example, if choose select Get Text > Object/Window and click on the Windows Clock application, a statement similar to the following is recorded in your test script:

```
# Clock settings 10:40:46 AM 8/8/95
win_get_text("Clock", text);
```

In the case of an object other than a window, the syntax is as follows:

obj_get_text (*object*, *text* **);**

The parameters of **obj_get_text** are identical to those of **win_get_text.**

Reading the Text from an Area of an Object or Window

The **win_get_text** and **obj_get_text** functions can be used to read text from a specified area of a window or other GUI object.

To read the text from an area of a window or object:

1 Choose Get Text > Area on the Create menu. Alternatively, if you are recording in Analog mode, press the GET TEXT AREA softkey.

WinRunner is minimized to an icon and the recording of mouse and keyboard input stops. The mouse pointer becomes a crosshairs pointer.

2 Use the crosshairs pointer to enclose the text to be read within a rectangle. Move the mouse pointer to one corner of the text you want to capture. Click and hold down the left mouse button. Drag the mouse until the rectangle encompasses the entire text, then release the mouse button. Click the right mouse button to capture the string.

You can preview the string before you capture it. Click the right mouse button before you release the left mouse button. (If your mouse has three buttons, release the left mouse button after drawing the rectangle and then press the middle mouse button.) The string appears under the rectangle or in the upper left corner of the screen.

WinRunner generates a **win_get_text** statement with the following syntax in the test script:

win_get_text (*window*, *text, x1,y1,x2,y2* **);**

For example, if you choose Get Text > Area and use the crosshairs to enclose only the date in the Windows Clock application, a statement similar to the following is recorded in your test script:

```
win_get_text ("Clock", text, 38, 137, 166, 185); # 8/13/95
```

The *window* is the name of the window. The *text* is an output variable that holds all of the captured text. *x1,y1,x2,y2* define the location from which to read text, relative to the specified window. When the function is recorded, the captured text is also inserted into the script as a comment.

The comment occupies the same number of lines in the test script as the text being read occupies on the screen. For example, if three lines of text are read, the comment will also be three lines long.

You can also read text from the screen by programming the Analog TSL function **get_text** into your test script. For more information, refer to the *TSL Online Reference*.

Note: When you read text with a learned font, WinRunner reads a single line of text only. If the captured text exceeds one line, only the leftmost line is read. If two or more lines have the same left margin, then the bottom line is read. See **Teaching Fonts to WinRunner** on page 263 for more information.

Searching for Text

You can search for text on the screen using the following TSL functions:

- The **win_find_text**, **obj_find_text, and find_text** functions determine the location of a specified text string.
- The **obj_move_locator_text**, **win_move_locator_text**, and **move_locator_text functions** move the mouse pointer to a specified text string**.**
- The **win_click_on_text, obj_click_on_text**, and **click_on_text** functions move the pointer to a string and click it.

Note that you must program these functions in your test scripts. You can use the Function Generator to do this, or you can type the statements into your test script.

Find

?

Getting the Location of a Text String

The **win_find_text** and **obj_find_text** functions perform the opposite of **win_get_text** and **obj_get_text**. Whereas the **get_text** functions retrieve any text found in the defined object or window, the **find_text** functions look for a specified string and return its location, relative to the window or object.

The **win_find_text** and **obj_find_text** functions are Context Sensitive and have similar syntax, as shown below:

win_find_text (*window, string***,** *result_array* [,x_1,y_1,x_2,y_2] [,*string_def*]);

obj_find_text (*object, string*, *result_array* [,x_1,y_1,x_2,y_2] [,*string_def*]);

The *window* or *object* is the name of the window or object within which WinRunner searches for the specified text. The *string* is the text to locate. The *result_array* is the name you assign to the four-element array that stores the location of the string. The optional x_1,y_1,x_2,y_2 specifies the region of the screen that is searched. If these parameters are not defined, WinRunner treats the entire window or object as the search area. The optional *string_def* defines how WinRunner searches for the text.

The **win_find_text** and **obj_find_text** functions return 1 if the search fails and 0 if it succeeds.

In the following example, **win_find_text** is used to determine where the total appears on a graph object in a Flight Reservation application.

```
set_window ("Graph", 10);
win_find_text ("Graph", "Total Tickets Sold:", result_array, 640,480,366,284, False);
```

You can also find text on the screen using the Analog TSL function **find_text**.

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

Note: When **win_find_text**, **obj_find_text**, or **find_text** is used with a learned font, then WinRunner searches for a single, complete word only. This means that any regular expression used in the *string* must not contain blank spaces, and only the default value of *string_def*, FALSE, is in effect.

Moving the Pointer to a Text String

The **win_move_locator_text** and **obj_move_locator_text** functions searches for the specified text string in the indicated window or other object. Once the text is located, the mouse pointer moves to the center of the text.

The **win_move_locator_text** and **obj_move_locator_text** functions are Context Sensitive and have similar syntax, as shown:

win_move_locator_text (*window, string,* [,x_1,y_1,x_2,y_2] [,*string_def*]);

obj_move_locator_text (*object, string,* [,x_1,y_1,x_2,y_2] [,*string_def*]);

The *window* or *object* is the name of the window or object that WinRunner searches. The *string* is the text to which the mouse pointer moves. The optional *x1,y1,x2,y2* parameters specify the region of the window or object that is searched. The optional *string_def* defines how WinRunner searches for the text.

In the following example, **obj_move_locator_text** moves the mouse pointer to a topic string in a Windows on-line help index.

```
function verify_cursor(win,str)
{
    auto text,text1,rc;
    # Search for topic string and move locator to text. Scroll to end of document,
    # retry if not found.
    set_window (win, 1);
    obj_mouse_click ("MS_WINTOPIC", 1, 1, LEFT);
     type ("<kCtrl_L-kHome_E>");
     while(rc=obj_move_locator_text("MS_WINTOPIC",str,TRUE)){
        type ("<kPgDn_E>");
        obj_get_text("MS_WINTOPIC", text);
        if(text==text1)
            return E_NOT_FOUND;
    text1=text;
    }
 }
```

You can also move the mouse pointer to a text string using the TSL Analog function **move_locator_text**. For more information on **move_locator_text**, refer to the *TSL Online Reference.*

Clicking a Specified Text String

The **win_click_on_text** and **obj_click_on_text** functions search for a specified text string in the indicated window or other GUI object, move the screen pointer to the center of the string, and click the string.

The **win_click_on_text** and **obj_click_on_text** functions are Context Sensitive and have similar syntax, as shown:

win_click_on_text (*window, string,* [,x_1,y_1,x_2,y_2] [,*string_def*] [,*mouse_button*]);

The *window* or *object* is the window or object to search. The *string* is the text the mouse pointer clicks. The optional *x1,y1,x2,y2* parameters specify the region of the window or object that is searched. The optional *string_def* defines how WinRunner searches for the text. The optional *mouse_button* specifies which mouse button to use.

In the following example, **obj_click_on_text** clicks a topic in an on-line help index in order to jump to a help topic.

```
function show_topic(win,str)

{
    auto text,text1,rc,arr[];
    # Search for the topic string within the object. If not found, scroll down to end
    # of document.
    set_window (win, 1);
    obj_mouse_click ("MS_WINTOPIC", 1, 1, LEFT);
     type ("<kCtrl_L-kHome_E>");
     while(rc=obj_click_on_text("MS_WINTOPIC",str,TRUE,LEFT)){
            type ("<kPgDn_E>");
            obj_get_text("MS_WINTOPIC", text);
            if(text==text1)
                return E_GENERAL_ERROR;
            text1=text;
            }
    }
```

For information on the **click_on_text** functions, refer to the *TSL Online Reference.*

Comparing Text

The **compare_text** function compares two strings, ignoring any differences that you specify. You can use it alone or in conjunction with the **win_get_text** and **obj_get_text** functions.

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

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

The *str1* and *str2* parameters represent the literal strings or string variables to be compared.

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

The **compare_text** function returns 1 when the compared strings are considered the same, and 0 when the strings are considered different. For example, a portion of your test script compares the text string “File” returned by **get_text**. Because the lowercase “l” character has the same shape as the uppercase “I”, you can specify that these two characters be ignored as follows:

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

Teaching Fonts to WinRunner

You use the Learn Font utility only when WinRunner cannot automatically read the text used by your application. In this case, you must teach your application's fonts to WinRunner.

To teach fonts to WinRunner, you perform the following main steps:

1 Use the Fonts Expert tool to have WinRunner learn the set of characters (fonts) used by your application.

2 Create a font group that contains one or more fonts.

   A *font group* is a collection of fonts that are bound together for specific testing purposes. Note that at any time, only one font group may be active in WinRunner. In order for a learned font to be recognized, it must belong to the active font group. However, a learned font can be assigned to multiple font groups.

3 Use the TSL **setvar** function to activate the appropriate font group before using any of the text functions.

Note that all learned fonts and defined font groups are stored in a *font library*. This library is designated by the XR_GLOB_FONT_LIB parameter in the *wrun.ini* file; by default, it is located in the *WinRunner installation directory/ fonts* subdirectory.

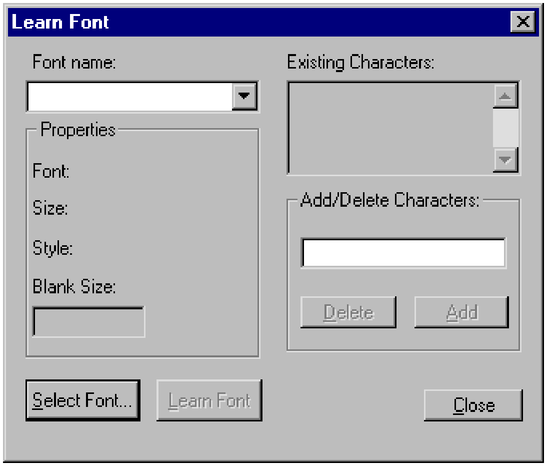
Learning a Font

If WinRunner cannot read the text in your application, use the Font Expert to learn the font.

To learn a font:

1 Choose Fonts Expert on the Tools menu or choose Programs > WinRunner > Fonts Expert on the Start menu. The Fonts Expert opens.

2 Choose Learn on the Font menu. The Learn Font dialog box opens.



3 Type in a name for the new font in the Font Name box (maximum of eight characters, no extension).

4 Click Select Font. The Font dialog box appears.

5 Choose the font name, style, and size on the appropriate lists.

6 Click OK.

7 Click Learn Font.

When the learning process is complete, the Existing Characters box displays all characters learned and the Properties box displays the properties of the fonts learned. WinRunner creates a file called *font_name.mfn* containing the learned font data and stores it in the font library.

8 Click Close.

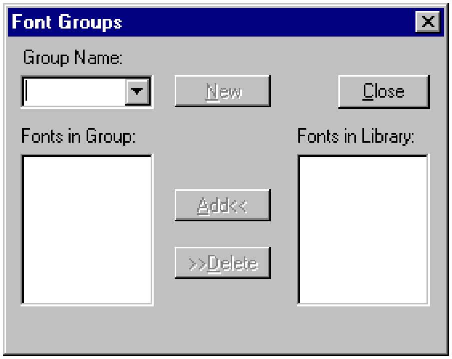
Creating a Font Group

Once a font is learned, you must assign it to a font group. Note that the same font can be assigned to more than one font group.

Note: Put only a couple of fonts in each group, because text recognition capabilities tend to deteriorate as the number of fonts in a group increases.

To create a new font group:

1 In the Fonts Expert, choose Groups on the Font menu. The Font Groups dialog box opens.



2 Type in a unique name in the Group Name box (up to eight characters, no extension).

3 In the Fonts in Library list, select the name of the font to include in the font group.

4 Click New. WinRunner creates the new font group. When the process is complete, the font appear in the Fonts in Group list.

WinRunner creates a file called *group_name.grp* containing the font group data and stores it in the font library.

To add fonts to an existing font group:

1. In the Fonts Expert, choose Groups on the Font menu. The Font Groups dialog box opens.
2. Select the desired font group from the Group Name list.
3. In the Fonts in Library list, click the name of the font to add.
4. Click Add.

To delete a font from a font group:

1. In the Fonts Expert, choose Groups on the Font menu. The Font Groups dialog box opens.
2. Select the desired font group from the Group Name list.
3. In the Fonts in Group list, click the name of the font to delete.
4. Click Delete.

Designating the Active Font Group

The final step before you can use any of the text functions is to activate the font group that includes the fonts your application uses.

To designate the active font:

1 Choose Options on the Settings menu.

The Options dialog box opens.

2 Click the Text Recognition tab.

3 In the Font Group box, enter a font group.

4 Click OK to save your selection and close the dialog box.

Only one group can be active at any time. By default, this is the group designated by the XR_FONT_GROUP system parameter in the *wrun.ini* file. However, within a test script you can activate a different font group or the **setvar** function together with the *fontgrp* test option.

For example, to activate the font group named editor from within a test script, add the following statement to your script:

```
setvar ("fontgrp", "editor");
```

For more information about choosing a font group from the Options dialog box, see Chapter 33, **Setting Global Testing Options**. For more information about using the **setvar** function to choose a font group from within a test script, see Chapter 34, **Setting Testing Options from a Test Script**.

Synchronizing Test Execution

Synchronization compensates for inconsistencies in the performance of your application during a test run. By inserting a synchronization point in your test script, you can instruct WinRunner to suspend the test run and wait for a visual cue to be displayed before resuming execution.

This chapter describes:

- **Waiting for Window and Object Bitmaps**
- **Waiting for Area Bitmaps**
- **Waiting for Property Values**

About Synchronizing Test Execution

Applications do not always respond to user input at the same speed from one test run to another. This is particularly common when testing applications running over a network. A synchronization point in your test script tells WinRunner to suspend test execution until the application being tested is ready, and then to continue the test.

For example, suppose that when testing a drawing application you want to import a bitmap from a second application and then rotate it. A human user would know to wait for the bitmap to be fully redrawn before trying to rotate it. WinRunner, however, requires a synchronization point in the test script after the import command and before the rotate command. Each time the test is run, the synchronization point tells it to wait for the import command to be completed before rotating the bitmap.

You can synchronize your test on a window or a GUI object in your application, or on any rectangular area of the screen that you select. To create a synchronization point, you choose the Wait Bitmap command on the Create menu and indicate an area or an object in the application being tested. WinRunner inserts a synchronization point in the script, captures an image of the specified bitmap or object, and stores it in the expected results directory (*exp*). The synchronization point appears as a **win_wait_bitmap** or **obj_wait_bitmap** statement in the test script. When you run the test, WinRunner suspends test execution and waits for the expected bitmap to appear. It then compares the current *actual* bitmap with the *expected* bitmap captured earlier. When the bitmap appears, test execution resumes.

Note that when recording a test in Analog mode, you should press the WAIT BITMAP or the WAIT BITMAP AREA softkey to create a synchronization point. This prevents WinRunner from recording extraneous mouse movements. If you are programming a test, you can use the Analog TSL function **wait_window** to wait for a bitmap. For more information, refer to the *TSL Online Reference*.

You can also create a synchronization point that waits until a specified property value is detected in a GUI object. For example, you can have WinRunner wait five seconds for an OK button in your application to become enabled before it tries to click the button. To create this type of synchronization point, you add a TSL **wait_info function** (such as **button_wait_info**) to your test script. Additional **wait_info** functions are available for other object types.

Waiting for Window and Object Bitmaps

You can create a synchronization point that waits for a window or object bitmap to appear in the application being tested. During a test run, WinRunner suspends test execution until the specified bitmap is redrawn, and then compares the current bitmap with the expected one captured earlier. If the bitmaps match, test execution proceeds. In the event of a mismatch, WinRunner displays an error message (when the *mismatch_break* testing option is on). For more information about the *mismatch_break* testing option, see Chapter 34, **Setting Testing Options from a Test Script**. You can also set this testing option globally using the corresponding Break when Verification Fails option in the Run tab of the Options dialog box. For more information about setting this testing option globally, see Chapter 33, **Setting Global Testing Options**.

The method for synchronizing a test is identical for object and for window bitmaps. You start by choosing Wait Bitmap > Object/Window on the Create menu. As you pass the mouse pointer over your application, objects and windows flash. To select a window bitmap, you click the title bar or the menu bar of the desired window. To select the bitmap of an object, click the object.

If the window or object you capture has a name that varies from run to run, you can define a regular expression in its physical description in the GUI map. This instructs WinRunner to ignore all or part of the name. For more information, see Chapter 5, **Editing the GUI Map**, and Chapter 23, **Using Regular Expressions**.

During recording, when you capture an object in a window other than the active window, WinRunner automatically generates a **set_window** statement.

To insert a synchronization point for a window or object bitmap:

1 On the Create menu, choose Wait Bitmap > Window/Object. Alternatively, if you are recording in Analog mode, press the WAIT BITMAP softkey. The mouse pointer becomes a pointing hand.

2 Highlight the desired window or object by placing the mouse pointer on it.

3 Click the left mouse button to complete the operation. WinRunner captures the bitmap and generates a **win_wait_bitmap** or **obj_wait_bitmap** statement with the following syntax in the test script.

win_wait_bitmap (*window*, *image*, *time* **);**

obj_wait_bitmap (*object, image, time* **);**

For example, suppose that while working with the Flight Reservation application, you decide to insert a synchronization point in your test script.
If you point to the Date of Flight box, the resulting statement might be:

```
obj_wait_bitmap ("Date of Flight:", "Img5", 22);
```

For more information on **obj_wait_bitmap** and **win_wait_bitmap**, refer to the *TSL Online Reference.*

Note: The execution of **win_wait_bitmap** and **obj_wait_bitmap** is affected by the current values specified for the *delay*, *timeout* and *min_diff* testing options. For more information on these testing options and how to modify them, see Chapter 34, **Setting Testing Options from a Test Script**. You may also set these testing options globally, using the corresponding Delay for Window Synchronization box, Timeout for Checkpoints and CS Statements box, and Threshold for Difference between Bitmaps box in the Run tab of the Options dialog box. For more information about setting these testing options globally, see Chapter 33, **Setting Global Testing Options**.

Waiting for Area Bitmaps

You can use a synchronization point to have WinRunner wait for a selected area bitmap in your application. The selected area may be any size; it may be part of a single object or window, or it may intersect several objects or windows.

To select an area bitmap, you use a crosshairs mouse pointer to define a rectangle around the area. WinRunner defines the rectangle using the coordinates of its upper left and lower right corners. These coordinates are relative to the upper left corner of the object or window containing the selected area. If the area intersects several objects in a window, the coordinates are relative to the window. If the selected area intersects several windows, or is part of a window with no title (a popup menu, for example), the coordinates are relative to the entire screen, or the root window.

During a test run, WinRunner suspends test execution until the specified bitmap is displayed. It then compares the current bitmap with the expected bitmap. If the bitmaps match, the operation was successful and test execution proceeds. In the event of a mismatch, WinRunner displays an error message (when the *mismatch_break* testing option is on). For more information, refer to Chapter 34, **Setting Testing Options from a Test Script**. You may also set this option using the corresponding Break when Verification Fails check box in the Run tab of the Options dialog box. For more information about setting this testing option globally, see Chapter 33, **Setting Global Testing Options**.

To define a synchronization point for an area bitmap:

1 On the Create menu, choose Wait Bitmap > Area. Alternatively, if you are recording in Analog mode, press the WAIT WINDOW (AREA) softkey. The mouse pointer changes into a crosshairs pointer.

2 Mark the area to capture. Click the left mouse button and drag the mouse until a rectangle encloses the area. Release the mouse button.

3 Click the right mouse button to complete the operation. WinRunner captures the bitmap and generates a **win_wait_bitmap or obj_wait_bitmap** statement with the following syntax in your test script.

obj_wait_bitmap (*object*, *image*, *time*, *x*, *y*, *width*, *height* **);**
win_wait_bitmap (*window*, *image*, *time*, *x*, *y*, *width*, *height* **);**

For example, suppose you are updating an order in the Flight Reservation application. You have to synchronize the continuation of the test with the appearance of a message verifying that the order was updated. You insert a synchronization point in order to wait for an “Update Done” message to appear in the status bar.

WinRunner generates the following statement:

```
obj_wait_bitmap ("Update Done...", "Img7", 10);
```

For more information on **win_wait_bitmap** and **obj_wait_bitmap**, refer to the *TSL Online Reference.*

Note: The execution of **win_wait_bitmap** and **obj_wait_bitmap** is affected by the current values specified for the *delay*, *timeout* and *min_diff* testing options. For more information on these testing options and how to modify them, see Chapter 34, **Setting Testing Options from a Test Script**. You may also set these testing options globally, using the corresponding Delay for Window Synchronization box, Timeout for Checkpoints and CS Statements box, and Threshold for Difference between Bitmaps box in the Run tab of the Options dialog box. For more information about setting these testing options globally, see Chapter 33, **Setting Global Testing Options**.

Waiting for Property Values

You can create a synchronization point that instructs WinRunner to wait for a specified property value to appear in a GUI object. During a test run, WinRunner suspends test execution until the property value is detected and then continues the test. For example, you can have WinRunner wait for a button to become enabled or for an item to be selected from a list.

To create a synchronization point that waits for a property value, add one of the following functions to the test script:

button_wait_info	**obj_wait_info**	**statusbar_wait_info**
edit_wait_info	**scroll_wait_info**	**tab_wait_info**
list_wait_info	**spin_wait_info**	
menu_wait_info	**static_wait_info**	

For information about working with these functions, refer to the *TSL Online Reference*.

For example, **button_wait_info** has the following syntax:

button_wait_info (*button*, *property*, *value*, *time* **);**

The *button* is the name of the button. The *property* is any property that is used by the button object class. The *value* is the value that must appear before the test run can continue. The *time* is the maximum number of seconds WinRunner should wait, added to the *timeout* testing option.

Suppose while testing the Flight Reservation application you order a plane ticket by typing in passenger and flight information and clicking Insert. The application takes a few seconds to process the order. Once the operation is completed, you click Delete to delete the order.

In order for the test to run smoothly, a **button_wait_info** statement is needed in the test script. This function tells WinRunner to wait up to 10 seconds (plus the timeout interval) for the Delete button to become enabled. This ensures that the test does not attempt to delete the order while the application is still processing it. The following is a segment of the test script:

```
button_press ("Insert");
button_wait_info ("Delete","enabled",0,"10");
button_press ("Delete");
```

Handling Unexpected Events and Errors

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

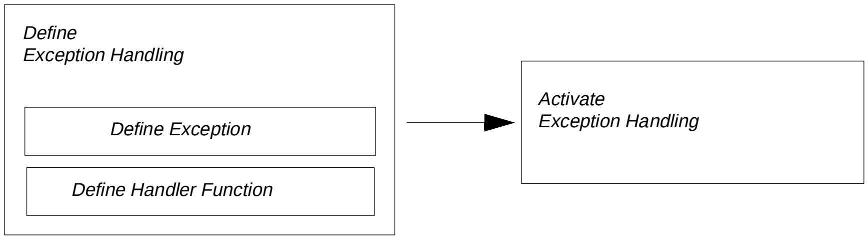
This chapter describes:

About Handling Unexpected Events and Errors

Unexpected events and errors 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.

Using *exception handling,* you can instruct WinRunner to detect an unexpected event when it occurs and act to recover the test run. For example, you can instruct WinRunner to detect a “Printer out of paper” message and call a handler function. The handler function recovers the test run by clicking the OK button to close the message.

To use exception handling, you must define and activate it.



Define exception handling: Describe the event or error you want WinRunner to detect, and define the actions it will perform in order to resume test execution. To do this, you define the exception and define a handler function.

Activate exception handling: Instruct WinRunner to look for any occurrence of the exception you defined.

Normally, you define and activate exception handling using the Exceptions dialog box. Alternatively, you can program exception handling in your test script using TSL statements. Both methods are described in this chapter.

WinRunner enables you to handle the following types of exceptions:

- **Pop-up exceptions:** Instruct WinRunner to detect and handle the appearance of a specific window.
- **TSL exceptions**: Instruct WinRunner to detect and handle TSL functions that return a specific error code.
- **Object exceptions:** Instruct WinRunner to detect and handle a change in a property for a specific GUI object.

Find
?

Handling Pop-Up Exceptions

Test execution is often disrupted by a window that pops up unexpectedly during a test run, such as a message box indicating that the printer is out of paper. Sometimes, test execution cannot continue until you close the window.

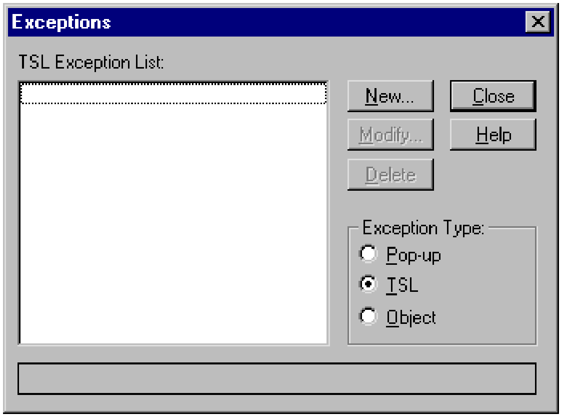
A pop-up exception instructs WinRunner to detect a specific window that may appear during a test run and to recover test execution, for example, by clicking an OK button to close a window.

Defining Pop-Up Exceptions

You use the Pop-Up Exception dialog box to define pop-up exceptions.

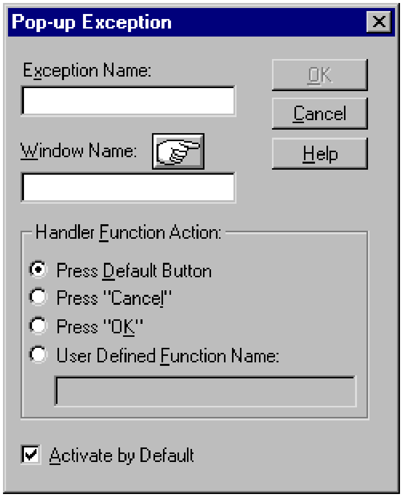
To define a pop-up exception:

1 Choose Exception Handling on the Tools menu to open the Exceptions dialog box.



Exception Type: Pop-up [✓ TSL] Object

2 Click Pop-Up in the Exception Type box. Click New to open the Pop-Up Exception dialog box.



Handler Function Action: [✓ Press Default Button] Press "Cancel" Press "OK" User Defined Function Name:

Activate by Default [✓]

3 In the Exception Name box, type in a new name.

4 Choose the pop-up window in one of the following ways:

Click the pointing hand and click the window. If the window to which you pointed is not in the GUI map, WinRunner adds it to the map. WinRunner enters the logical name into the Window Name box.

Type the name of the window into the Window Name box. You can type in the window's title or its logical name. If the window is not in the GUI map, WinRunner assumes that the name you specify is the window's title. You can also specify a regular expression.

5 Choose a handler function: click a default (press the Enter key, click OK, or click Cancel), or click User Defined Function Name to specify a user-defined handler. The dialog box changes to display the user-defined Handler Function Name box.

If you specify a user-defined handler function that is undefined or in an unloaded compiled module, the Handler Function Definition dialog box opens automatically, displaying a handler function template. For more information on defining handler functions, see **Defining Handler Functions for Pop-Up Exceptions** on page 287.

6 To activate exception handling active at all times, select the Activate by Default check box.

7 Click OK to complete the definition and close the dialog box. The new exception appears in the Pop-Up Exception List in the Exceptions dialog box.

WinRunner activates handling and adds the new exception to the list of default pop-up exceptions in the Exceptions dialog box. Default exceptions are defined by the XR_EXCP_POPUP configuration parameter in the *wrun.ini* file.

As an alternative to using the Pop-Up Exception dialog box, you can define a pop-up exception in a test script using the **define_popup_exception** function, and you can activate it using the **exception_on** function. (For more information on activating and deactivating exceptions, see **Activating and Deactivating Exception Handling** on page 303.) Note that exceptions you define using TSL are valid only for the current WinRunner session. For more information on **define_popup_exception**, refer to the *TSL Online Reference*.

Defining Handler Functions for Pop-Up Exceptions

The handler function is responsible for recovering test execution. When WinRunner detects a specific window, it calls the handler function. You implement this function to respond to the unexpected event in a way that meets your specific testing needs.

When defining an exception from the Pop-Up Exception dialog box, you can specify one of two types of handler functions:

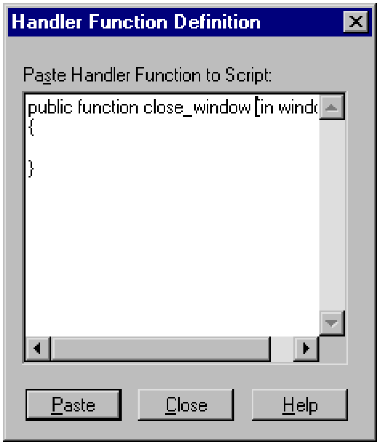
- **Default actions:** WinRunner clicks the OK or Cancel button in the pop-up window, or presses Enter on the keyboard. To select a default handler, click the appropriate button in the dialog box.
- **User-defined handler:** If you prefer, specify the name of your own handler. Click User Defined Function Name and type in a name in the User Defined Function Name box.

If you specify a user-defined handler that is either undefined or in an unloaded compiled module, WinRunner automatically displays a template in the Handler Function Definition dialog box. You can use the template to help you create a handler function. The handler function must receive the *window_name* as a parameter.

To define your own handler function using the Handler Function Definition dialog box:

1 Define an exception using the Pop-Up Exception dialog box. Specify a new name for the handler function.

2 Click OK. The dialog box closes and the Handler Function Definition dialog box opens, displaying the handler function template.


Handler Function Definition
Paste Handler Function to Script:
Paste
Close
Help

```
public function close_window (in windc
{

}
```

3 Create a function that closes the window and recovers test execution.

4 Click Paste to paste the statements into the WinRunner editor.

5 Click Close to close the Handler Function Definition dialog box.

6 You can edit the test script further if necessary. When you are done, save the script in a compiled module.

User-defined handler functions should be stored in a compiled module. For WinRunner to call the function, the compiled module must be loaded when the exception occurs. For more information, refer to Chapter 20, **Creating Compiled Modules**.

In the following example, the handler function is edited to handle an error message. A Flights Reservation application sometimes displays a “FATAL DATABASE ERROR” message, often as the result of a faulty database entry. You can create a handler function that gets the faulty entry number and its value, and writes the information to the test execution report. Then, it dismisses the error message.

The script segment below shows how the handler function (my_win_handler) can be edited in the template:

```
public function my_win_handler(string win_name)
{
    auto order_num;
    set_window("Open Order",2);
    edit_get_text("Order Value",order_num);
    report_msg("Database error. Order number:" & order_num);
    button_press (“OK);
}
```

Find

?

Handling TSL Exceptions

A TSL exception enables you to detect and respond to a specific error code returned during test execution.

Suppose you are running a batch test on an unstable version of your application. If your application crashes, you want WinRunner to recover test execution. A TSL exception can instruct WinRunner to recover test execution by exiting the current test, restarting the application, and continuing with the next test in the batch.

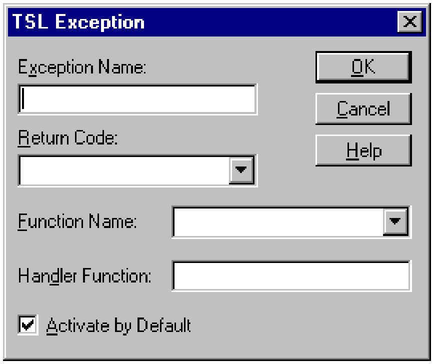
Defining TSL Exceptions

You use the TSL Exception dialog box to define, modify, and delete TSL exceptions.

To define a TSL exception:

1 Choose Exception Handling on the Tools menu to open the Exceptions dialog box.

2 Click TSL in the Exception Type box and click New to open the TSL Exception dialog box.



3 In the Exception Name box, type in a new name.

4 In the Return Code list, choose a return code.

5 In the Function Name list, choose a TSL function. If you choose <<any function>> or do not specify a function, WinRunner defines the exception for any TSL function that returns the specified return code.

6 In the Handler Function box, type in the name of a handler function.

   If you specify a handler function that is undefined or is in an unloaded compiled module, the Handler Function Definition dialog box opens automatically, displaying a handler function template. For more information on defining handler functions, see **Defining Handler Functions for TSL Exceptions** on page 293.

7 To activate exception handling at all times, select the Activate by Default check box.

8 Click OK to complete the definition and close the dialog box. The new exception appears in the TSL Exception List in the Exceptions dialog box.

Once you have defined the exception, WinRunner activates handling and adds the exception to the list of default TSL exceptions in the Exceptions dialog box. Default TSL exceptions are defined by the XR_EXCP_TSL configuration parameter in the *wrun.ini* configuration file.

As an alternative to using the TSL Exception dialog box, you can define a TSL exception in a test script using the **define_TSL_exception** function, and you can activate it using the **exception_on** function. (For more information on activating and deactivating exceptions, see **Activating and Deactivating Exception**

Handling on page 303.) Note that exceptions you define using TSL are valid only for the current WinRunner session. For more information on **define_TSL_exception**, refer to the *TSL Online Reference*.

Defining Handler Functions for TSL Exceptions

The handler function is responsible for recovering test execution. When WinRunner detects a specific error code, it calls the handler function. You implement this function to respond to the unexpected error in the way that meets your specific testing needs.

If you specify a handler that is either undefined or in an unloaded compiled module, WinRunner automatically displays a template in the Handler Function Definition dialog box. You can use the template to help you create a handler function. The handler function must receive the *return_code* and the *function_name* as parameters.

To define a handler function using the Handler Function Definition dialog box:

1 Define an exception using the TSL Exception dialog box. Specify a new name for the handler function.

2 Click OK. The dialog box closes and the Handler Function Definition dialog box opens, displaying the handler function template.

3 Create a function that recovers test execution.

4 Click Paste to paste the statements into the WinRunner window. Click Close to close the Handler Function Definition dialog box.

5 You can further edit the test script if necessary. When you are done, save the script in a compiled module.

In order for the exception to call the handler function, the compiled module must be loaded when the exception occurs. For more information, refer to Chapter 20, **Creating Compiled Modules**.

The following example uses the Flight Reservation application to demonstrate how you can instruct WinRunner to record a specific event in the test report. In the application, it is illegal to select an item from the “Fly To:” list without first selecting an item from the “Fly From:” list.

Suppose you program a stress test to create such a situation. The test selects the first item in the “Fly From:” list for every selection from the “Fly To:” list. If the “Fly From:” list is empty, the command:

```
list_select_item ("Fly From:","#0");
```

returns the error code E_ITEM_NOT_FOUND.

You could implement exception handling to identify each occurrence of the E_ITEM_NOT_FOUND return value for the **list_select_item** command. You do this by defining a handler function that reacts by recording the event in the test report.

Edit the handler function (list_item_handler) in the template as follows:

```
public function list_item_handler(rc, func_name)
{
report_msg("List Fly From: is empty")
}
```

Note: The handler function of a TSL exception does not need to return any value. However, a TSL exception defined for a TSL Context Sensitive function can return one of the following values:

- RETRY: The function is executed again. If the exception recurs, it is not handled again. An exception handler should return RETRY if the problem that caused the exception is resolved.
- DEF_PROCESSING: The function is handled by default, as though no exception was defined. The TSL command that called the exception is processed as though an exception was never detected (i.e. messages are generated, the Run wizard opens, or the return value is reported).

For example, if a **button_press** statement returns a value of E_NOT_UNIQUE, and this error code is defined as an exception, the exception handler is called. If it returns DEF_PROCESSING, the Run wizard opens and tries to resolve the problem of the non-unique button. Therefore, an exception handler should return DEF_PROCESSING when the handler cannot resolve the exception.

Handling Object Exceptions

During testing, unexpected changes can occur to GUI objects in the AUT. These changes are often subtle but they can disrupt test execution and distort results.

One example is a change in the color of a button. 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.

You could use exception handling to detect a different color in the button during the test run, and to recover test execution by calling a handler function that closes the network and restores the button's color.

Find

?

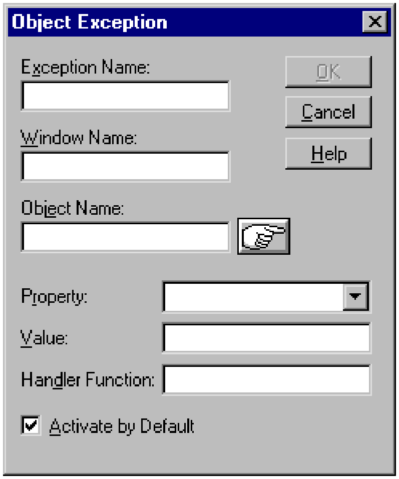
Defining Object Exceptions

You use the Object Exception dialog box to define, modify, and delete object exceptions.

To define an object exception:

1 Choose Exception Handling on the Tools menu to open the Exceptions dialog box.

2 Click Object in the Exception Type box and click New to open the Object Exception dialog box.



3 In the Exception Name box, type in a new name.

Find
?

4 Choose the object in one of the following ways:

   Click the pointing hand and click the object. The names of the object and its parent window appear in the boxes.

   Type the names of the object and its parent window. In the Object Name box, type in the name of the object. In the Window Name box, type in the name of the window in which the object is found.

   Note that for an object exception, the object and its parent window must be in the loaded GUI map when exception handling is activated.

5 In the Property list, choose the property for which you are defining the object exception.

6 In the Value box, type in a value for the property you have selected. If you do not specify a value, the exception will be defined for any change from the current property value.

   Note that the property you specify for the exception cannot appear in the object’s physical description. If you attempt to specify such a property, WinRunner will display an error message. To work around the problem, modify the object’s physical description.

7 In the Handler Function box, enter the name of the handler function.

   If you specify a handler function that is undefined or in an unloaded compiled module, the Handler Function dialog box opens, displaying a handler function template. For more information on defining handler functions, see **Defining Handler Functions for Object Exceptions** on page 300.

8 To make exception handling active at all times, select the Activate by Default check box.

If you have not specified a value for the property, ensure that the object is displayed when you press the OK button. You can activate exception handling only if WinRunner can learn the current value of the property.

9 Click OK to complete the definition and close the dialog box. The new exception appears in the Object Exception List in the Exceptions dialog box.

Once you have defined the exception, WinRunner activates handling and adds the exception to the list of default object exceptions in the Exceptions dialog box. Default object exceptions are defined by the XR_EXCP_OBJ configuration variable in the wrun.ini file.

As an alternative to using the Object Exception dialog box, you can define an object exception in a test script using the **define_object_exception** function, and you can activate it using the **exception_on** function. (For more information on activating and deactivating exceptions, see **Activating and Deactivating Exception Handling** on page 303.) Note that exceptions you define with TSL are valid only for the current WinRunner session. For more information on **define_object_exception**, refer to the *TSL Online Reference*.

Defining Handler Functions for Object Exceptions

The handler function is responsible for recovering test execution. When WinRunner detects a changed property, it calls the handler function. You implement this function to respond to the unexpected event in a way that meets your specific testing needs.

If you specify a handler function that is either undefined or in an unloaded compiled module, WinRunner automatically displays a template in the Handler Function Definition dialog box. You can use the template to help you create a handler function. The handler function must receive the *window, object, property* and *value* as parameters*.*

Note that the first command in the template is **exception_off.** This is because an object exception does not detect the actual change in the specified object property; rather, it detects a state in the specified object property. The handler function must deactivate exception handling as soon as the function begins to execute. If not, the exception will immediately reoccur, calling the handler function endlessly.

Only if the handler function has fixed the problem that caused the exception to occur, call **exception_on** at the bottom of the handler function so that if the exception reoccurs, it will be detected again. (Note that the **exception_on** statement appears in the the template, but it is commented out.)

To define a handler function using the Handler Function Definition dialog box:

1 Define an exception using the Object Exception dialog box.

2 Click OK. The dialog box closes and the Handler Function Definition dialog box opens, displaying the handler function template.

3 Create a function that recovers test execution.

4 Click Paste to paste the statements into WinRunner. The dialog box closes.

5 Click Close to exit the Handler Function Definition dialog box.

6 You can further edit the test script if necessary. When you are done, save the script in a compiled module. To enable exception detection, ensure that you load the compiled module before test execution.

Handler functions must be stored in a compiled module. For WinRunner to call the handler function, the compiled module must be loaded at the moment the exception occurs. For more information, refer to Chapter 20, **Creating Compiled Modules**.

For example, the labels of GUI objects may become corrupted during testing, often as a result of memory management problems. You could define exception handling to take care of such irregularities in the Flights application.

Find
?

The handler function that is called might write the unexpected event to a test report, close and restart your application, then exit the current test and continue to the next test in the batch. To do this, you would edit the handler function (label_handler) in the template as follows:

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

Activating and Deactivating Exception Handling

When you define an exception by using the Exceptions dialog box, exception handling is activated by default. To turn off activating exception handling by default, clear the Activate by Default check box in each Exception dialog box.

You can also activate exception handling in a test script by using TSL commands:

- To instruct WinRunner to begin detecting an exception, insert an **exception_on** statement at the appropriate point in your test script.
- To instruct WinRunner to stop detecting an exception, use the **exception_off** function. Use **exception_off_all** to stop detection of all active exceptions.

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

Programming with TSL

Programming with TSL
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:

Find

?

About Enhancing Your Test Scripts

When you record a test, a test script is generated in Mercury Interactive’s Test Script Language (TSL). Each TSL statement in the test script represents keyboard and/or mouse input to the application being tested.

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. TSL is easy to use because you do not have to compile. You enhance a recorded test script simply by typing programming elements into the test window, 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 17, **Using Visual Programming**.

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 on TSL, refer to the *TSL Online Reference*.

Find

?

Statements

When WinRunner records a test, each line it generates in the test script is a statement. A statement is any expression that is followed by a semicolon. A single statement may be longer than one line in the test script.

For example:

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

If you program a test script by typing directly into the test window, remember to include a semicolon at the end of each statement.

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.

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.

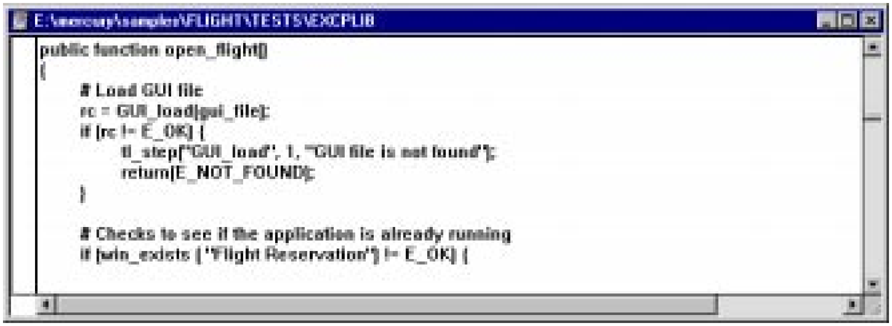
For example:

```
# 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");
```

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) {
```

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 a Report** on page 320. For information about variable and constant declarations, see Chapter 19, **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 (unary)
-	subtraction (binary)
*	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 Online 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, *statement* is executed and *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
```

Find

?

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);
```

Find

?

Decision-Making

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 *expression1* 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.

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

# 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);
}
```

Find

?

Sending Messages to a Report

You can define a message in your test script and have WinRunner send it to the test report. To send a message to a report, 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 report 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 36, **Initializing Special Configurations**, for more information.

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:\\flight1a.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 completed, 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 TestDirector 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 *TestDirector 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 directory, then the full path must be given. The optional *save_file* parameter indicates the name of a third file in which both compared files are stored.

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 the new file is identical to the original one and to store both files in the test directory.

```
# Open win.ini using Notepad.
system("write.exe win.ini");
set_window("Document - NotePad",1);

# Save win.ini as win1.ini
menu_select_item("File;Save As...");
set_window("Save As");
edit_set("File Name:_0","d:\Win95\win1.ini");
set_window("Save As", 10);
button_press("Save");

# Compare win.ini to win1.ini and save both files to "save".
file_compare("d:\\win95\\win.ini","d:\\win95\\win1.ini","save");
```

For information on viewing the results of file comparison, see Chapter 26, **Analyzing Test Results**.

Programming with TSL
Using Visual Programming

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

This chapter describes:

Find

?

About Visual Programming

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:

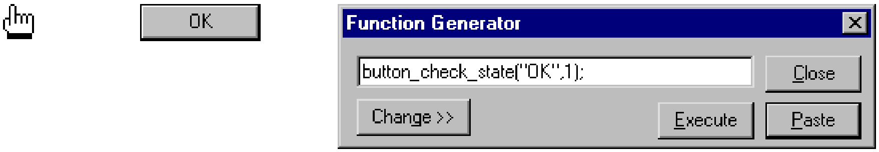
- Add Context Sensitive functions that perform operations on a GUI object or get information from the application being tested.
- Add Standard and Analog functions that perform non-Context Sensitive tasks such as synchronizing test execution or sending user-defined messages to a report.
- Add Customization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

Find
?

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

button_check_state("OK",1);



```
button_check_state("OK",1);
```

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

To change to another function for the object, click Change. Once you have generated a statement, you can use it in two different ways, separately or together:

- *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 added automatically to the temporary GUI map file when the generated statement is executed or pasted into the test script.

Note that 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 35, **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 339.

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 > Object/Window on the Create menu. WinRunner shrinks to an icon and the mouse pointer becomes a pointing hand.

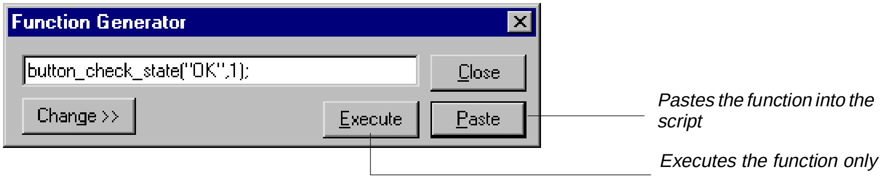
2 Point to a GUI object in the application being tested. Each object flashes as you pass the mouse pointer over it.

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 but is not pasted into the test script.


Function Generator
Close
Change >>
Execute
Paste
Pastes the function into the script
Executes the function only

```
button_check_state("OK",1);
```

5 Click Close to close the dialog box.

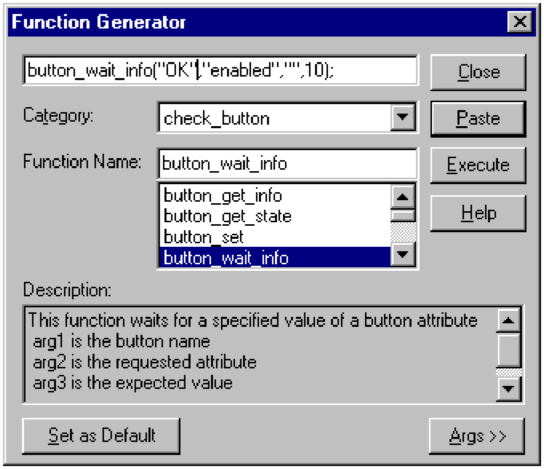
Selecting a Different Function for a GUI Object

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 different function for a GUI object:

1. Choose Insert Function > Object/Window on the Create menu. WinRunner is minimized 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 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 the Change button. 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. Select a function from the Function Name list. 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","",10);

Category: check_button

Function Name: button_wait_info

button_get_info
button_get_state
button_set
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 336 to learn 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 List on the Create menu to open the Function Generator dialog box.
2. Select a function category from the Category list. 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. Choose a function from the Function Name list. 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 336 to learn how to fill in the argument text boxes.

Find
?

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.

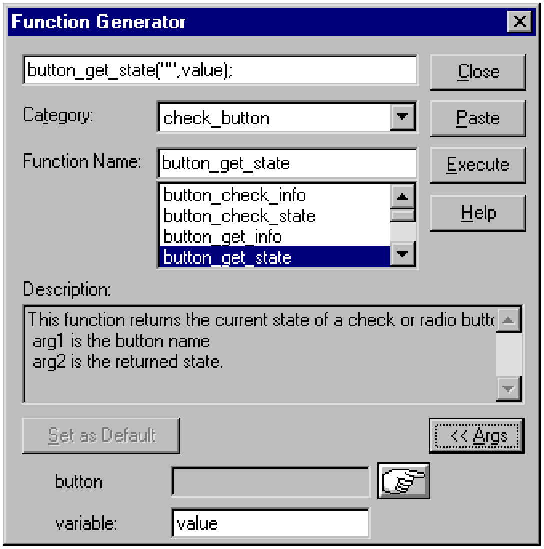
Assigning Argument Values

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 In the Function Generator dialog box, select a category and a function name.

2 Click Args. The dialog box expands based on the number of arguments in the function.



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 butt
arg1 is the button name
arg2 is the returned state.

button

variable: value

3 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 states “The current function cannot be applied to the pointed object.” 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_info**. 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 Open the Function Generator dialog box and select a function category.

2 Select the function that you want to make the default from the Function Name list.

3 Click Set as Default.

4 Click Close.

The selected function remains the default function in its category until it is changed or until you end your WinRunner session. To save changes to the default function setting, add a **generator_set_default_function** statement to your

Find
?

startup test. For more information on startup tests, see Chapter 36, **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.

Programming with TSL
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

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, passes parameter values, and controls test execution.

When WinRunner interprets a **call** statement in a test script, it opens and runs the called test. 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. 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.

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 27, **Running Batch Tests**.

Using the Call Statement

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

call *test_name* **([** *parameter1*, *parameter2*, ...*parametern* **]);**

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

Any called test must be stored in a directory 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. To do so, choose Calls on the Debug menu.

Returning to the Calling Test

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

- 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. If **treturn** or **texit** is not used, or if no value is specified, then the return value of the **call** statement is 0.

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 the help_test is also returned to the

Find
?

batch test, and is assigned to the variable *return_val*. If **texit** is not used, *return_val* is 0.

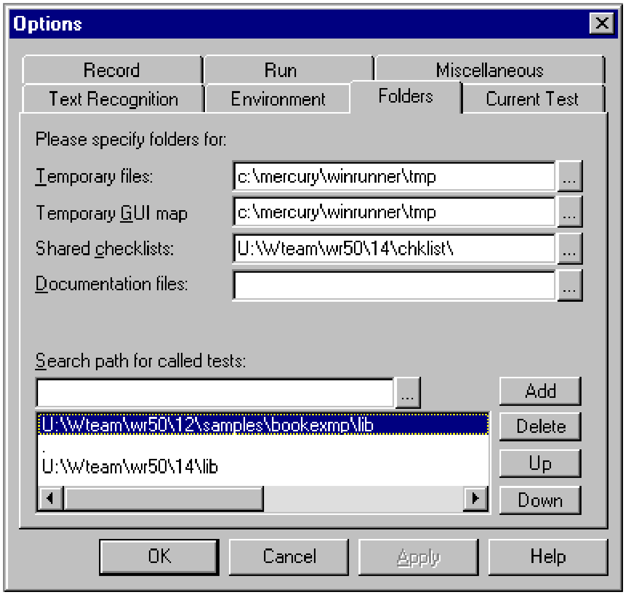
For more information on batch tests, see Chapter 27, **Running Batch Tests**.

Setting the Search Path

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

To set the search path, choose Options on the Settings menu. The Options dialog box opens. Click the Folders tab 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.

- To add a folder to the search path, type in the directory name in the text box. Use the Add, Up, and Down buttons to position this folder in the list.
- To delete a search path, select its name from the list and click Delete.



Please specify folders for:

Temporary files: c:\mercury\winrunner\tmp

Temporary GUI map: c:\mercury\winrunner\tmp

Shared checklists: U:\Wteam\wr50\14\chklist\

Documentation files:

Search path for called tests:

U:\Wteam\wr50\12\samples\bookexmp\lib
.
U:\Wteam\wr50\14\lib

For more information about how to set a search path in the Options dialog box, see Chapter 33, **Setting Global Testing Options**.

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 the current test run only.

For example:

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

This statement tells WinRunner to search the c:\ui_tests directory for called tests. For details on **setvar**, see Chapter 34, **Setting Testing Options from a Test Script**.

Note: If WinRunner is connected to TestDirector, you can also set a search path within a TestDirector database. For more information, see **Using TSL Functions with TestDirector** on page 715.

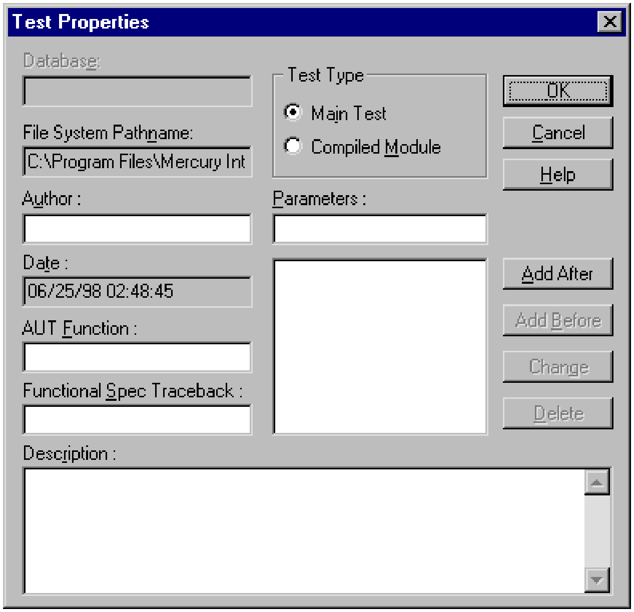
Defining Test Parameters

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

For example, suppose you define two 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. Subsequently, two values supplied by a calling test will supply the x- and y-coordinates of the mouse pointer.

Defining Test Parameters in the Test Properties Dialog Box

Parameters are defined in the Test Properties dialog box. To open this dialog box, choose Test Properties on the File menu.



To define a new parameter:

1 In the Test Properties dialog box, type the name of the parameter in the Parameters box.

2 Select a parameter from the list and click either Add After or Add Before.

Note that since parameter values are assigned sequentially, the order in which parameters are listed determines the value that is assigned to a parameter by the calling test.

3 Click OK to close the dialog box.

To delete a parameter from the parameter list:

1 In the Test Properties dialog box, select the name of the parameter to delete.

2 Click Delete.

3 Click OK to close the dialog box.

To modify the name of a parameter:

1 In the Parameter list of the Test Properties dialog box, select the parameter to modify.

2 Type in the new name in the Parameters box.

3 Click Change.

4 Click OK to close the dialog box.

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);
print(i);  # prints "5"

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

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);
print(a[1]); # prints "104"

# 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 *TSL Online Reference.*

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

```
# test a, with parameter k
i = 1;
j = 2;
k = 3;
call test b(i);
print(j & k & l); # prints '256

# test_b, with parameter j
j = 4;
k = 5;
l = 6;
print (i & j & k); # prints '145'
```

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 User-Defined Functions

In addition to providing built-in functions, TSL allows you to design and implement your own functions. You can:

- Create user-defined functions in a test script. You define the function once, and then call it from anywhere in the test (including called tests).
- Create user-defined functions in a compiled module. Once you load the module, you can call the functions from any test. For more information, see Chapter 20, **Creating Compiled Modules**.
- Call functions from the Microsoft Windows API or any other external functions stored in a DLL. For more information, see Chapter 21, **Calling Functions from External Libraries**.

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.

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;
}

Find
?

Class

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

Once you execute a public function, it is available to all tests, for as long as the test containing the function remains open. 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. The functions in a compiled module are available for the duration of the testing session.

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

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);
```

Illegal usage of the function call would be

get_date (date[i]); **or** get_date ("Today's date is"& todays_date);

because both contain expressions.

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.

Return Statement

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.

Find

?

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. Additional information about declarations can be found in the *TSL Online Reference.*

Variables

Variable declarations have the following syntax:

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

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

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. 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 20, **Creating Compiled Modules**.

Find

?

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.)

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);
```

Find

?

```
#Compare label to file name
if ( file != lbl)
    return 1;
else
    return 0;
}
rc=open_file("c:\\dash\\readme.tx");
pause(rc);
```

Creating Compiled Modules

Compiled modules are libraries of frequently-used functions. You can save user-defined functions in compiled modules and then call the functions from your test scripts.

This chapter describes:

Find

?

About Compiled Modules

A compiled module is a script containing a library of user-defined functions that you want to call frequently from other tests. When you load a compiled module, its 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
- 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, your tests will require less error-checking. In addition, calling a function that is already compiled is significantly faster than interpreting a function in a test script.

You can compile a module in one of two ways:

- Run the module script using the WinRunner Run commands.
- Load the module from a test script using the TSL **load** function.

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.

You can add **load** statements to your startup test. This ensures that the functions in your compiled modules are automatically compiled each time you start WinRunner. See Chapter 36, **Initializing Special Configurations**, for more information.

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 by clicking Compiled Module in the Test Type box in the Test Properties dialog box (see **Creating a Compiled Module** on page 374).

The content of a compiled module differs from that of an ordinary test: 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 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 19, **Creating User-Defined Functions**.
- prototypes of external functions. For more information, see Chapter 21, **Calling Functions from External Libraries**.
- **load** statements to other modules. For more information, see **Loading and Unloading a Compiled Module** on page 377.

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

- A public 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 or public) are initialized.

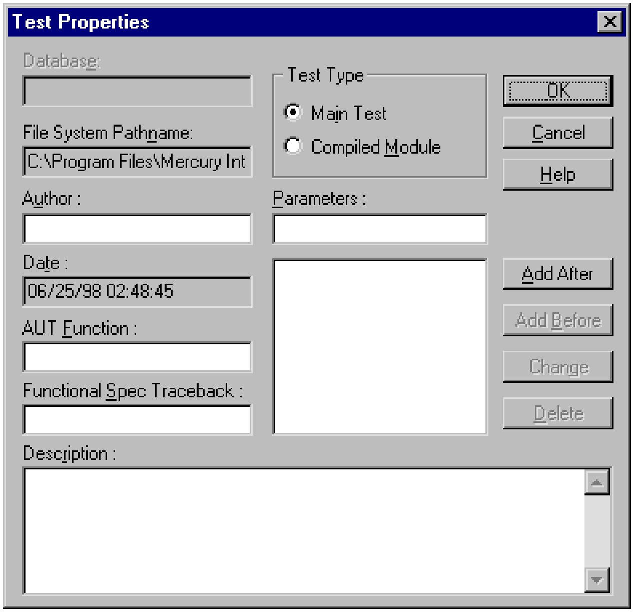
Creating a Compiled Module

Creating a compiled module is similar to creating a regular test script.

To create a compiled module:

1. Open a new test.
2. Write the user-defined functions.
3. Choose Test Properties on the File menu to open the Test Properties dialog box.

4 Click Compiled Module and click OK.


Test Properties

Database:

File System Pathname: C:\Program Files\Mercury Int

Author :

Date : 06/25/98 02:48:45

AUT Function :

Functional Spec Traceback :

Description :

Test Type
[✓ Main Test] Compiled Module

Parameters :

5 Choose Save on the File menu.

Save your modules in a location that is readily available to all your tests. When a module is loaded, WinRunner locates it according to the search path you define. For more information on defining a search path, see Chapter 18, **Calling Tests**.

6 Compile the module using the **load** function. See **Loading and Unloading a Compiled Module** below for more information.

Loading and Unloading a Compiled Module

In order to access the functions in a compiled module you need to load the module. You can load it from within any test script using the **load** command; all tests will then be able to access the function until you quit WinRunner or unload the compiled module.

If you create a compiled module that contains frequently-used functions, you can load it from your startup test. For more information, see Chapter 36, **Initializing Special Configurations**.

You can load a module either as a *system* module or as a *user* module. A system module is generally a closed module that is “invisible” to the tester. It is not displayed when it is loaded, cannot be stepped into, and is not stopped by a pause command. A system module is not unloaded when you execute an **unload** statement with no parameters (global unload).

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

Find

?

load

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

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

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

Two additional, optional parameters indicate the type of module. The first parameter indicates whether the function 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)

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.

unload

The **unload** function removes 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.

reload

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* [,1|0] [,1|0]);

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

Two additional optional parameters indicate the type of module. The first parameter indicates whether the module is a system module or a user module: 1 indicates a system module; 0 indicates a user module.

(Default = 0)

The second optional parameter indicates whether 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 a module, use **unload** followed by **load**, or else use the **reload** function.

If you try to load a module that has already been loaded, WinRunner does not load it again. Instead, it initializes variables and increments a *load counter*. If a module has been loaded several times, then the **unload** statement does not unload the module, but rather decrements the counter. For 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. WinRunner does not unload the function; it decrements the load counter. When execution returns to test A, *math_functions* is still loaded.

Incremental Compilation

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

To perform incremental compilation:

1 Open the module.

2 Choose Test Properties on the File menu.

3 In the Test Properties dialog box, click Main Test in the Test Type Options box. Then click OK.

4 To load an entire module, choose Run from Top on the Run menu.

To incrementally compile part of a module, run the necessary statements using the Step command.

5 Choose Test Properties on the File menu.

6 In the Test Properties dialog box, click Compiled Mode in the Test Type Options box to change the test type back. Then click OK.

7 If necessary, save the module and close it.

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;
}
```

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

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 link to DLLs that are part of the application being tested in order to access its exported functions.

Using the **load_dll** function, you dynamically link with 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: The Windows API functions appear by default in WinRunner's Function Generator under the WinAPI category. For information on using the Function Generator, see Chapter 17, **Using Visual Programming**. For information about specific Windows API functions, refer to your *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** directory in the installation directory.

Dynamically Linking with External Libraries

In order to link to 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 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");
```

The **load_16_dll** function performs a runtime load of a 16-bit DLL. It has the following syntax:

load_16_dll (*pathname* **);**

The *pathname* is the full pathname of the 16-bit DLL to be loaded.

Find
?

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 DLL to be unloaded.

To unload all loaded DLLs from memory, use the following statement:

```
unload_dll ("");
```

The **unload_16_dll** function unloads a loaded external 16-bit DLL. It has the following syntax:

unload_16_dll (*pathname* **);**

The *pathname* is the full pathname of the 16-bit DLL to be unloaded.

To unload all loaded DLLs from memory, use the following statement:

```
unload_16_dll ("");
```

For more information, refer to the *TSL Online 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 36, **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*)
long (signed and unsigned)	*unsigned int*

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.

In addition, your exported 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 exported 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 *GetWindowText* 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 directory)
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 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 directory)
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);
```

Creating Dialog Boxes for Interactive Input

WinRunner enables you to create dialog boxes that you can use to pass input to your test during interactive test execution.

This chapter describes:

Find

?

About Interactive Test Input

You can create dialog boxes that pop up during interactive test execution, 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 interactive test execution, 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 interactive execution.
- **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 interactive execution.
- **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 interactive execution, 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 test execution, 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 test execution 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:

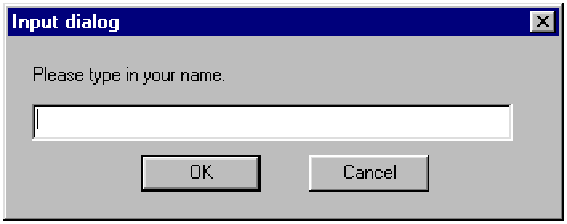
create_input_dialog (*message* **);**

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.");
```


Input dialog

Please type in your name.

The input that is typed into the dialog box during test execution 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*.

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.

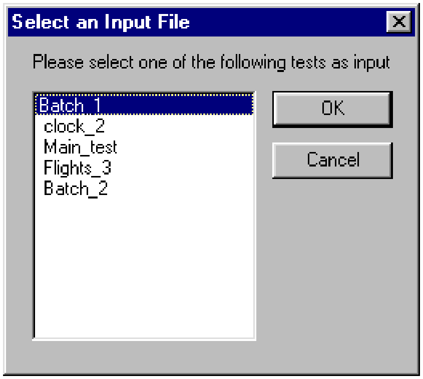
You use the TSL function **create_list_dialog** to create a list dialog box. This function has the following syntax:

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 test execution 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).

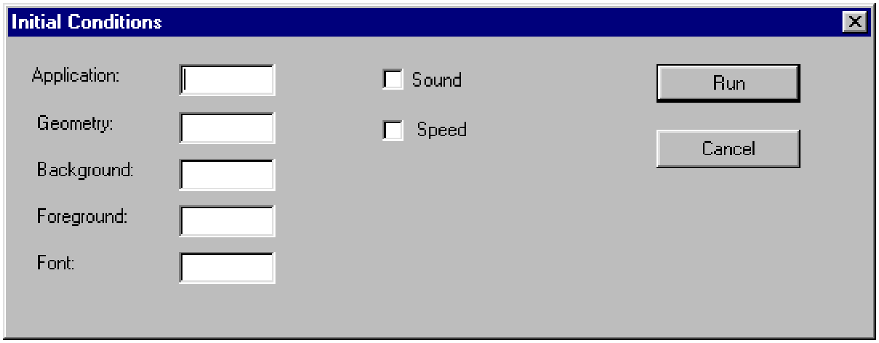
- *check_name* contains the labels of the check boxes in the dialog box. Multiple check box labels are separated by commas, and all the labels together are considered a single string. If the dialog box has no check boxes, this parameter must be an empty string (empty quotation marks).

  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:

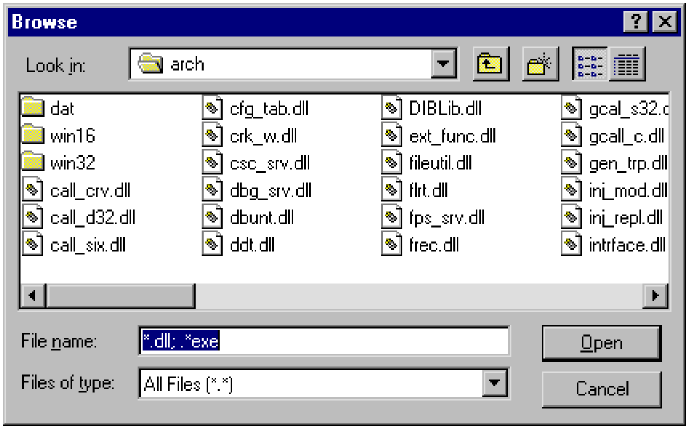
create_browse_file_dialog (*filter* **);**

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.).

Find
?

In the following example, the browse dialog box displays all files with extensions .dll or .exe.

filename = create_browse_file_dialog("*.dll;*.exe");



When the OK 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 interactive test execution is saved to variables for analysis.

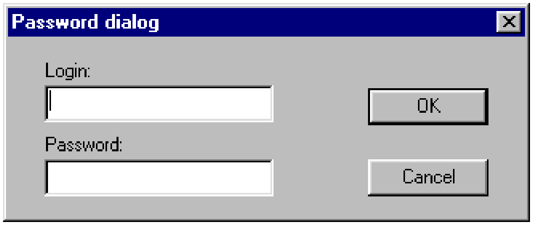
You use the TSL function **create_password_dialog** to create a password dialog box. This function has the following syntax:

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 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 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.

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

- **When to Use Regular Expressions**
- **Regular Expression Syntax**

About Regular Expressions

Regular expressions allow 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 which specifies a complex search phrase. In most cases the string is preceded by an exclamation point (!). 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 **Regular Expression Syntax** on page 414 for more information.

Note that WinRunner regular expressions include options similar to those offered by the UNIX grep command.

When to Use Regular Expressions

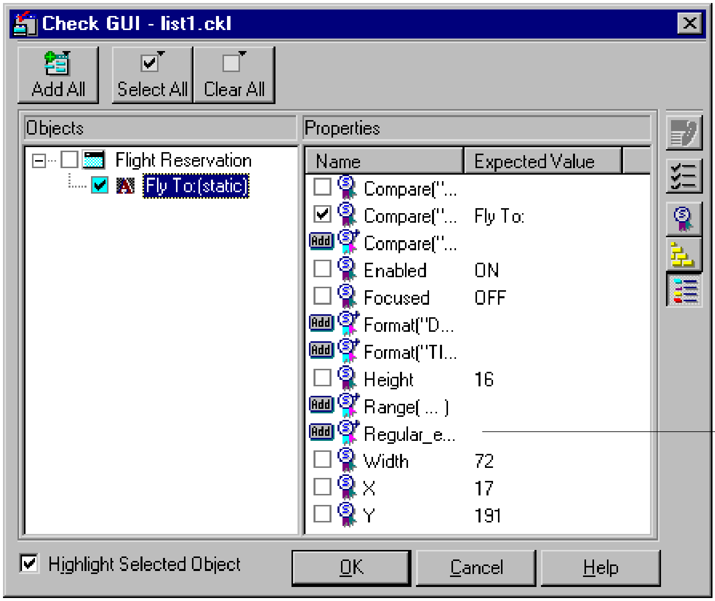
Use a regular expression when the name of a GUI object can vary each time you run a test. For example, you can use a regular expression:

- 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 pushbutton if its label toggles from "Start" to "Stop".

- In a GUI checkpoint, when evaluating the contents of an edit object or static text object with a varying name. You define the regular expression by creating a custom check for the object. For example, if you choose a Check GUI command on the Create menu and double-click a static text object, you can define a regular expression in the Check GUI dialog box:



Flight Reservation
Fly To:(static)

Name	Expected Value
Compare("...	
Compare("...	Fly To:
Compare("...	
Enabled	ON
Focused	OFF
Format("D...	
Format("TI...	
Height	16
Range(...)	
Regular_e...	
Width	72
X	17
Y	191

Highlight Selected Object

Clicking the Add button defines a GUI check on a static text object with varying text.

Note that 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.

- In a text checkpoint, to locate a varying text string using **win_find_text** or **object_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, see Chapter 5, **Editing the GUI Map**.

Regular Expression Syntax

Regular expressions must begin with an exclamation point (!), except when defined in a Check GUI dialog box or in a text checkpoint. 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.

The following options 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.

Matching One or More Specific Characters

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

```
Q*
```

causes WinRunner to match Q, QQ, QQQ, etc. For example, in the following physical description, the regular expression enables WinRunner to locate any pushbutton that starts with “O” (for example, On or Off).

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

The above statement uses two special characters: “.” and “*”. Since the asterisk follows the period, WinRunner locates any combination of characters. You can also use a combination of brackets and an asterisk to limit the label to a combination of non-numeric characters only:

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

Find
?

Programming with TSL
Data-Driven Tests

WinRunner enables you to create and run a test script in which data stored in an external table drives the test.

This chapter describes:

Find

?

About Data-Driven Tests

When you test your application, you may want to check how it performs the same operations on different sets of data. For example, suppose you want to check how your application responds to ten separate sets of data. You could record ten separate tests, each with its own set of data. Alternatively, you could create a *data-driven* test with a loop that runs ten times: each time the loop runs, it is driven by a different set of data. In order for WinRunner to use data to drive the test, you must link the data to the test script which it drives. This is called *parameterizing* your test. The data is stored in a *data table*. This chapter explains how to create data-driven tests in WinRunner.

To create a data-driven test, you first record a test script and insert checkpoints. You then perform the following steps:

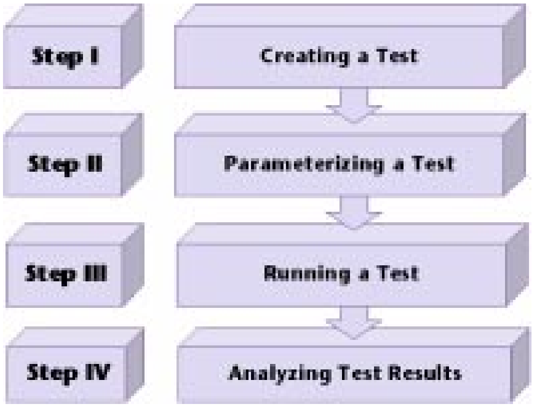
1 In your test script, replace specific values with parameters in checks and in recorded statements.

2 Create a data table with variable values for the parameters.

3 Add certain statements to your test script so that WinRunner reads from the data table and runs in a loop for each iteration of data.

The Data-Driven Testing Process

The following diagram outlines the stages of preparing and running data-driven tests in WinRunner:



Step I - Creating a Data-Driven Test

Creating a test involves two tasks:

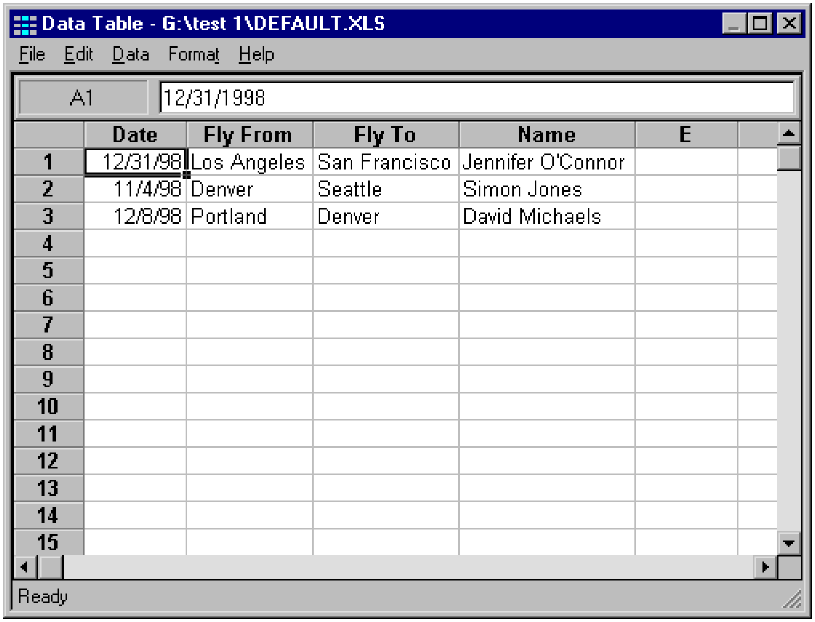
- You perform operations in your application. WinRunner records the actions that you perform. For information about recording tests, see Chapter 8, **Creating Tests**.
- While recording operations on your application, you insert checkpoints at the appropriate places. WinRunner performs the checks when you run the test. For information about adding GUI to your test scripts, see Chapter 9, **Checking GUI Objects**, Chapter 12, **Checking Bitmaps**, and Chapter 13, **Checking Text**.

Step II - Parameterizing a Test

Once you have created a basic test that uses a single set of data, you are ready to parameterize your test. When you parameterize your test, you replace fixed values in your test script with parameters. A parameter is a variable that is assigned a value from outside the test in which it is defined.

Parameterization enables you to use a single test to conduct checks on your application with multiple sets of data. You can parameterize any variable information in your application, including recorded statements and checks.

In addition to inserting parameters in your test script, you must also create a data table in which to store the variable data. When you run your test, WinRunner replaces the parameters in your test script with the fixed values from the data table. In each iteration during your test, WinRunner changes the values in the parameterized statements.


Data Table - G:\test 1\DEFAULT.XLS
File Edit Data Format Help
A1
Ready

12/31/1998

	Date	Fly From	Fly To	Name	E
1	12/31/98	Los Angeles	San Francisco	Jennifer O'Connor	
2	11/4/98	Denver	Seattle	Simon Jones	
3	12/8/98	Portland	Denver	David Michaels	
4					
5					
6					
7					
8					
9					
10					
11					
12					
13					
14					
15					

- Each row in the data table generally represents the values that WinRunner submits for all the parameterized fields during a single iteration of the test. For example, a loop in a test that is associated with a table with ten rows will run ten times.
- Each column in the table represents the list of values for a single parameter during all the iterations of a test.

Before you are ready to run your data-driven test, you must also add the following types of statements to it:

- statements to open and close the data table, where the data is stored
- statements to run the test script in a loop while there is a row of data in the data table with which to drive the test script

Step III - Running a Test

After you create a data-driven test, you run it as you would run any other WinRunner test. WinRunner uses the data in the data table in place of the parameters in your test script. While it runs, WinRunner opens the data table and conducts the checks that you defined. For more information on running a test, see Chapter 25, **Running Tests**.

Step IV - Analyzing Test Results

When a test run is complete, you can view the test results as you would for any other WinRunner test. WinRunner generates and displays a report that shows details of every check conducted during the test.

If you inserted a **ddt_report_row** statement in your test script, then WinRunner prints a row of the data table to the test results. Each iteration of a **ddt_report_row** statement in your test script creates a line in the Test Results window in which Table Row appears in the Event column. Double-clicking this line displays all the parameterized data used by WinRunner in an iteration of the test. For more information on the **ddt_report_row** function, see **Reporting the Active Row in a Data Table to the Test Results** on page 450 or refer to the *TSL Online Reference*. For information on viewing test results, see Chapter 26, **Analyzing Test Results**.

Find
?

Parameterizing a Data-Driven Test

There are three main steps to creating a data-driven test:

1. Record on the application you are testing and insert checks to create a basic test.
2. Replace specific values in checks and in recorded statements with parameters, and create a data table with variable values for the parameters.
3. Add certain statements and functions to your test so that it will read from the data table and run in a loop while it reads each iteration of data.

Creating a Basic Test

You start creating a data-driven test by recording a test, as usual, with one set of data. In the following example, the user wants to check that the Insert Order button is enabled once flight information is entered for a variety of passengers. The test is recorded using a specific passenger's flight data.

For example, in the Flight Reservations application, you create an order by entering the flight information and the passenger's name. Then you click Insert Order. A test script similar to the following is created:

```
# Activate the Flight Reservation window.
set_window ("Flight Reservation", 23);

# Enter the flight date.
edit_set ("Date of Flight:", "12/31/98");

# Enter the flight origin and destination.
list_select_item ("Fly From:", "Los Angeles");  # Item Number 1;
list_select_item ("Fly To:", "San Francisco");  # Item Number 2;

# Select a flight.
obj_mouse_click ("FLIGHT", 41, 31, LEFT);
set_window ("Flights Table", 2);
button_press ("OK");
set_window ("Flight Reservation", 19);

# Enter the passenger's name.
edit_set ("Name:", "Jennifer O'Connor");

# Click the Insert Order button.
button_press ("Insert Order");

# Insert a synchronization point that waits for the order to be inserted.
obj_wait_bitmap("Insert Done...", "Img1", 15);

# Open a new order.
menu_select_item ("File;New Order");
```

The purpose of this test is to check that the Insert Order button is enabled once the flight and passenger information has been entered and before clicking the Insert Order button. Normally you would insert an **obj_check_gui** statement while recording your test to check that this button is enabled. However, all **_check_gui** statements contain references to checklists, and a checklist cannot be parameterized in a data-driven test. Instead you use a **button_check_info** statement to check that the Insert Order button is enabled. You can insert the following statements into your test script:

```
if (button_check_info("Insert Order","enabled",TRUE) != E_OK)
    report_msg("Insert Order button not enabled.");
```

When WinRunner uses the **button_check_info** statement to check a property of a button, it does not use a checklist. Similarly, all **_check_info** statements check properties of GUI objects without using a GUI checklist. In the example above, if the Insert Order button is not enabled, WinRunner sends a message to the test results. Refer to the *TSL Online Reference* for more information about these functions.

Parameterizing Values in a Test Script and Creating a Data Table

You create a data-driven test by parameterizing arguments within the test and creating a data table in which data is stored.

In the example in **Creating a Basic Test** on page 425 there are four statements that contain fixed values entered by the user:

```
edit_set ("Date of Flight:", "12/31/98");

list_select_item ("Fly From:", "Los Angeles");  # Item Number 1;

list_select_item ("Fly To:", "San Francisco");  # Item Number 2;

edit_set ("Name:", "Jennifer O’Connor");
```

You parameterize the statements that contain data by replacing this data with parameters.

To replace the data in your test script with parameters and create a data table:

1. Choose Tools > Data Table. The data table opens.

   Note that the name of the data table is Default.XLS. This data table is stored in the same folder as your test.

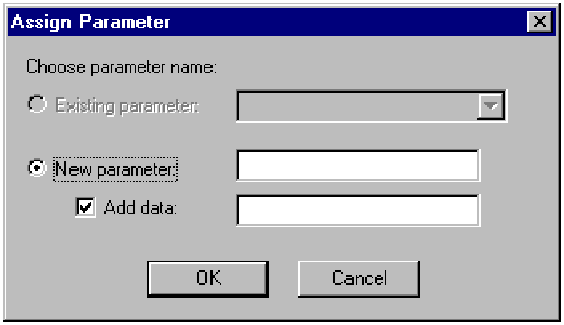
2. In your test script, select the first instance in which data that you want to parameterize appears. For example, in the first **edit_set** statement in the test script above, select:

   "12/31/98"

Note: Make sure that you select the quotes surrounding the data when you select the data.

---

3 Choose Tools > Parameterize Value. The Assign Parameter dialog box opens.



4 Click New Parameter. Enter the name for the parameter in the New Parameter box.

For example, for the following statement, enter “Date” (without quotation marks) into the New Parameter box.

```
edit_set ("Date of Flight:", "12/31/98");
```

The data which was selected in your test script, with quotation marks, appears, without quotation marks in the Add Data box.

5 Click OK.

In the test script, the data selected in the test script is replaced with a **ddt_val** statement which contains the name of the table and the name of the parameter you created in the data table.

In the example, the date "12/31/98" is replaced with a **ddt_val** statement which contains the name of the table and the parameter “Date”, so that the original statement appears as follows:

```
edit_set ("Date of Flight:", ddt_val(table, "Date"));
```

In the data table, a new column is created with the name of the parameter you assigned.

In the example, a new column is created with the header Date.

Note: If you do not want to include the data currently selected in the test script in the data table, clear the Add Data check box.

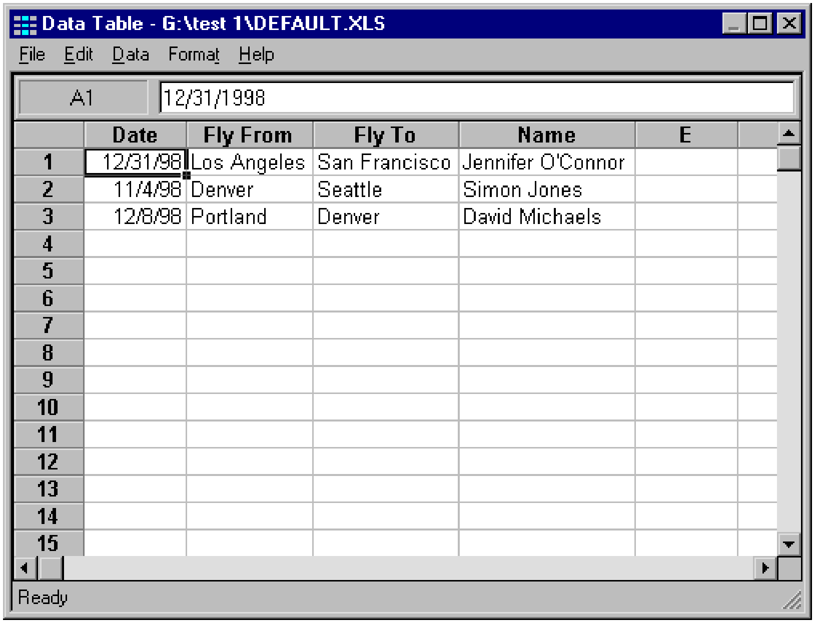
Note: You can also assign the data to an existing parameter. To assign the data to a column you already created in the data table, click Existing Parameter. Select an existing parameter from the list.

Repeat steps 2 to 4 for each argument you want to parameterize.

6 Enter additional data into the data table manually.

Note: You can also import data from a Microsoft Excel 5.0 or 7.0 file or from a tabbed text file into the data table. To import a file, use the File > Import command in the data table. The file you import replaces any data currently in the data table.

The data table displays three sets of data that were entered for the test in the example. The first set of data was entered using the Tools > Parameterize Value command in WinRunner. The next two sets of data were entered into the data table manually.


Data Table - G:\test 1\DEFAULT.XLS
File Edit Data Format Help
Ready

A1 12/31/1998

	Date	Fly From	Fly To	Name	E
1	12/31/98	Los Angeles	San Francisco	Jennifer O'Connor	
2	11/4/98	Denver	Seattle	Simon Jones	
3	12/8/98	Portland	Denver	David Michaels	
4					
5					
6					
7					
8					
9					
10					
11					
12					
13					
14					
15					

- Each row in the data table generally represents the values that WinRunner submits for all the parameterized fields during a single iteration of the test. For example, a loop in a test that is associated with a table with ten rows will run ten times.

- Each column in the table represents the list of values for a single parameter during all the iterations of a test.

7 Select File > Close to close the data table.

When you close the data table, it is automatically saved with the corresponding WinRunner test script. Whenever you save a WinRunner test, the corresponding data table is saved automatically. Note that the data table viewer does not need to be open in order to run a data-driven test.

Adding Statements to Your Test Script to Open and Close the Data Table

Add the following statements to your test script immediately preceding the parameterized portion of the script:

```
table=getvar("testname") & "\\Default.xls";
rc=ddt_open (table=default.xls);
if (rc!=E_OK)
    pause("ERROR");
do
{
```

These statements open the data table for the test and run the statements between the curly brackets that follow for each row of data in the data table.

Add the following statements to your test script immediately following the parameterized portion of the script:

```
}
while (ddt_next_row (default.xls)==E_OK);
ddt_close (default.xls);
```

These statements run the statements that appear within the curly brackets above as long as there is data in the next row of the data table. They use the data from the next row of the data table to drive each successive iteration of the test. When the next row of the data table is empty, these statements stop running the statements within the curly brackets and close the data table.

Guidelines for Creating a Data-Driven Test

Consider the following guidelines when creating a data-driven test:

- A data-driven test can contain more than one parameterized loop.
- A data-driven test can be driven by more than one data table.
- You can change the active row during the test run by using TSL statements. For more information, see **Using TSL Functions with Data-Driven Tests** on page 446.
- You can read from a non-active row during the test run by using TSL statements. For more information, see **Using TSL Functions with Data-Driven Tests** on page 446.
- It is not necessary to use all the data in a data table when running a data-driven test.
- If you want, you can parameterize only part of your test script or a loop within it.
- If WinRunner cannot find a GUI object that has been parameterized while running a test, make sure that the parameterized argument is not surrounded by quotes in the test script.
- You can use the data table in the same way as an Excel spreadsheet. You can insert formulas into cells.
- It is not necessary for the data table viewer to be open when you run a test.

Editing the Data Table

The data table contains the values that WinRunner uses for parameterized input fields and checks when you run a test. You can edit information in the data table by typing directly into the table. You can use the data table in the same way as an Excel spreadsheet. You can insert formulas into cells.

To edit the data table:

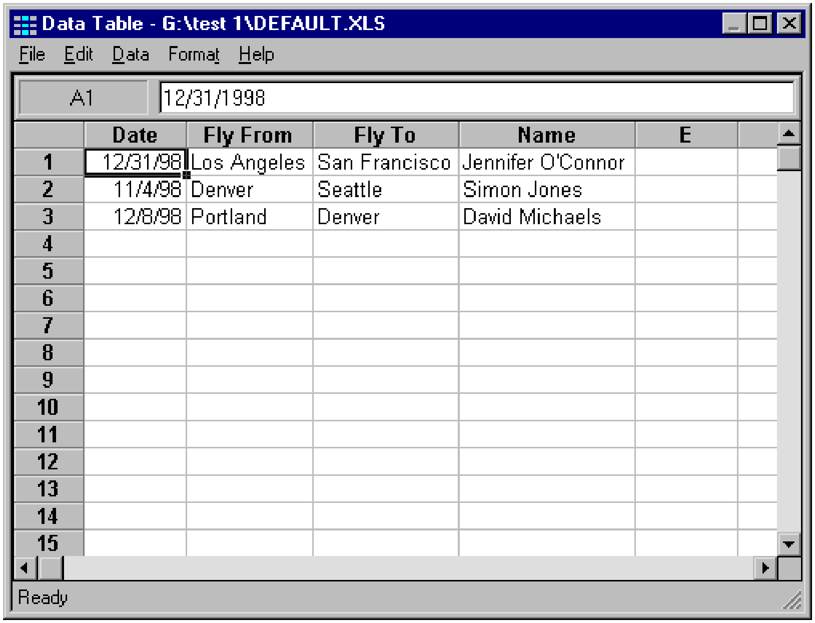
1 Open your test.

2 Choose Tools > Data Table.

Find

?

The data table for the test opens.


Data Table - G:\test 1\DEFAULT.XLS
File Edit Data Format Help
A1
Ready

12/31/1998

	Date	Fly From	Fly To	Name	E
1	12/31/98	Los Angeles	San Francisco	Jennifer O'Connor	
2	11/4/98	Denver	Seattle	Simon Jones	
3	12/8/98	Portland	Denver	David Michaels	
4					
5					
6					
7					
8					
9					
10					
11					
12					
13					
14					
15					

- Each row in the data table generally represents the values that WinRunner submits for all the parameterized fields during a single iteration of the test. A loop in a test that is associated with a table that has, for example, ten rows, will run ten times. There are TSL functions that enable you to change the active row or the active cell in the data table during a test run. For more information, see **Using TSL Functions with Data-Driven Tests** on page 446, or refer to the *TSL Online Reference*.

- Each column in the table represents the list of values for a single parameter during all the iterations of a test.

3 Use the menu commands described below to edit the data table.

4 Select File > Close to close the data table.

Find

?

File Menu

Use the File menu to import and export, close, save, and print the data table. WinRunner automatically saves the data table for a test in the test folder and names it *default.xls*. The following commands are available in the File menu:

File Command	Description
Import	Imports an existing table file into the data table. The file can be a Microsoft Excel 5.0 or 7.0 file or a tabbed text file. Note that the cells in the first row of an Excel file become the column headers in the data table viewer. Note that the new table file replaces any data currently in the data table.
Export	Saves the data table as a Microsoft Excel 5.0 or 7.0 file or as a tabbed text file. Note that the column headers in the data table viewer become the cells in the first row of an Excel file.
Close	Closes the data table.
Print	Prints the data table.
Print Preview	Previews how the data table will print.
Print Setup	Allows you to select the printer to which the data table is sent, the page orientation, and paper size.

Edit Menu

Use the Edit menu to move, copy, and find selected cells in the data table. The following commands are available in the Edit menu

Edit Command	Description
Cut	Cuts the data table selection writes it to the Clipboard.
Copy	Copies the data table selection to the Clipboard.
Paste	Pastes the contents of the Clipboard to the current data table selection.
Paste Values	Pastes values from the Clipboard to the current data table selection. Any formatting applied to the values is ignored. In addition, only formula results are pasted; formulas are ignored.
Clear All	Clears both the format of the selected cells, if the format was specified using the Format menu commands, and the values (including formulas) of the selected cells.
Clear Formats	Clears the format of the selected cells, if the format was specified using the Format menu commands. Does not clear values (including formulas) of the selected cells.
Clear Contents	Clears only values (including formulas) of the selected cells. Does not clear the format of the selected cells.
Insert	Inserts empty cells at the location of the current selection. Cells adjacent to the insertion are shifted to make room for the new cells.

Edit Command	Description
Delete	Deletes the current selection. Cells adjacent to the deleted cells are shifted to fill the space left by the vacated cells.
Copy Right	Copies data in the leftmost cell of the selected range to the right to fill the range.
Copy Down	Copies data in the top cell of the selected range down to fill the range.
Find	Finds a cell containing specified text. You can search by row or column in the table and specify to match case or find entire cells only.
Replace	Finds a cell containing specified text and replaces it with different text. You can search by row or column in the table and specify to match case or find entire cells only. You can also replace all.
Go To	Goes to a specified cell. This cell becomes the active cell.

Data Menu

Use the Data menu to recalculate formulas, sort cells and edit autofill lists. The following commands are available in the Data menu:

Data Command	Description
Recalc	Recalculates any formula cells in the data table.
Sort	Sorts a selection of cells by row or column and keys.
AutoFill List	Creates, edits or deletes an autofill list. An autofill list contains frequently-used series of text such as months and days of the week. When adding a new list, separate each item with a semi-colon. To use an autofill list, enter the first item into a cell in the data table. Drag the cursor across or down and WinRunner automatically fills in the cells in the range according to the autofill list.

Format Menu

Use the Format menu to set the format of data in a selected cell or cells. The following commands are available in the Format menu:

Format Command	**Description**
General	Sets format to General. General displays numbers with as many decimal places as necessary and no commas.
Currency(0)	Sets format to currency with commas and no decimal places.
Currency(2)	Sets format to currency with commas and two decimal places.
Fixed	Sets format to fixed precision with commas and no decimal places.
Percent	Sets format to percent with no decimal places. Numbers are displayed as percentages with a trailing percent sign (%).
Fraction	Sets format to fraction.
Scientific	Sets format to scientific notation with two decimal places.
Date: (M/d/yy)	Sets format to Date with the M/d/yy format.
h:mm AM/PM	Sets format to Time with the H:MM AM/PM format.

Format Command	Description
Custom Number	Sets format to a custom number format that you specify.
Validation Rule	Sets validation rule to test data entered into a cell or range of cells. A validation rule consists of a formula to test, and text to display if the validation fails.

Find

?

Using TSL Functions with Data-Driven Tests

WinRunner provides several TSL functions that enable you to work with data-driven tests.

You can use the Function Generator to insert the following functions in your test script, or you can manually program statements that use these functions. For information about working with the Function Generator, see Chapter 17, **Using Visual Programming**. For more information about these functions, refer to the *TSL Online Reference*.

Find

?

Opening a Data Table

The **ddt_open** function opens the specified data table. The data table is a Microsoft Excel 5.0 or 7.0 file or a tabbed text file. The first row in the Excel/tabbed text file contains the names of the parameters. This function has the following syntax:

ddt_open (*data_table_filename* **);**

The *data_table_filename* is the name of the data table file.

Closing a Data Table

The **ddt_close** function closes the specified data table. This function has the following syntax:

ddt_close (*data_table_filename* **);**

The *data_table_filename* is the name of the data table file.

Returning the Number of Rows in a Data Table

The **ddt_get_row_count** function returns the number of rows in the specified data table. This function has the following syntax:

ddt_get_row_count (*data_table_filename*, *out_rows_count* **);**

The *data_table_filename* is the name of the data table file. The *out_rows_count* is the output variable that stores the total number of rows in the data table.

Changing the Active Row in a Data Table to the Next Row

The **ddt_next_row** function changes the active row in the specified data table to the next row. This function has the following syntax:

ddt_next_row (*data_table_filename* **);**

The *data_table_filename* is the name of the data table file.

Setting the Active Row in a Data Table

The **ddt_set_row** function sets the active row in the specified data table. This function has the following syntax:

ddt_set_row (*data_table_filename*, *row* **);**

The *data_table_filename* is the name of the data table file. The *row* is the new active row in the data table.

Retrieving the Active Row of a Data Table

The **ddt_get_current_row** function retrieves the active row in the specified data table. This function has the following syntax:

ddt_get_current_row (*data_table_filename*, *out_row* **);**

The *data_table_filename* is the name of the data table file. The *out_row* is the output variable that stores the specified row in the data table.

Determining whether a Parameter in a Data Table is Valid

The **ddt_is_parameter** function determines whether a parameter in the specified data table is valid. This function has the following syntax:

ddt_is_parameter (*data_table_filename*, *parameter* **);**

The *data_table_filename* is the name of the data table file. The *parameter* is the name of the parameter in the data table.

Returning the Value of a Parameter in the Active Row in a Data Table

The **ddt_val** function returns whether the value of a parameter in the active row in the specified data table is valid. This function has the following syntax:

ddt_val (*data_table_filename*, *parameter* **);**

The *data_table_filename* is the name of the data table file. The *parameter* is the name of the parameter in the data table.

Returning the Value of a Parameter of a Row in a Data Table

The **ddt_val_by_row** function returns whether the value of a parameter in the specified row of the specified data table is valid. This function has the following syntax:

ddt_val_by_row (*data_table_filename*, *row_number*, *parameter* **);**

The *data_table_filename* is the name of the data table file. The *parameter* is the name of the parameter in the data table. The *row_number* is the number of the row in the data table.

Reporting the Active Row in a Data Table to the Test Results

The **ddt_report_row** function reports the active row in the specified data table to the test results. This function has the following syntax:

ddt_report_row (*data_table_filename* **);**

The *data_table_filename* is the name of the data table file.

Running Tests

Find

?

Running Tests

Once you have developed a test script, you run the test to check the behavior of your application.

This chapter describes:

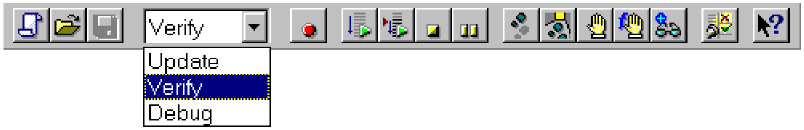
About Running Tests

When you run a test, WinRunner interprets your test script, line by line. The execution arrow in the left margin of the test script marks each TSL statement as it is interpreted. As the test runs, WinRunner operates your application as though a person were at the controls.

You can run your tests in three modes:

- Verify mode, to check your application
- Debug mode, to debug your test script
- Update mode, to update the expected results

You choose a run mode from the list of modes on the Standard toolbar. The Verify mode is the default run mode.



Use WinRunner's Run commands to run your tests. You can run an entire test, or a portion of a test. Before running a Context Sensitive test, make sure the necessary GUI map files are loaded. For more information, see Chapter 4, **Creating the GUI Map**.

You can run individual tests or use a batch test to run a group of tests. A batch test is particularly useful when your tests are long and you prefer to run them overnight or at other off-peak hours. For more information, see Chapter 27, **Running Batch Tests**.

WinRunner Test Execution Modes

WinRunner provides three modes in which to run tests—Verify, Debug, and Update. You use each mode during a different phase of the testing process.

Verify

Use the Verify mode to check your application. WinRunner compares the *current* response of your application to its *expected* response. Any discrepancies between the current and expected responses are captured and saved as *verification results*. When you finish running a test, by default the Test Results window opens for you to view the verification results. For more information, see Chapter 26, **Analyzing Test Results**.

You can save as many sets of verification results as you need. To do so, save the results in a new directory each time you run the test. You specify the directory name for the results using the Run Test dialog box. This dialog box opens each time you run a test in Verify mode. For more information about running a test script in Verify mode, see **Running a Test to Check Your Application** on page 461.

Find

?

Debug

Use the Debug mode to help you identify bugs in a test script. Running a test in Debug mode is the same as running a test in Verify mode, except that debug results are always saved in the *debug* directory. Because only one set of debug results is stored, the Run Test dialog box does not open automatically when you run a test in Debug mode.

When you finish running a test in Debug mode, the Test Results window does not open automatically. To open the Test Results window and view the debug results, you can click the Test Results button on the main toolbar or choose the Test Results command on the Tools menu.

Once you run a test in Debug mode, it remains the default run mode for the current WinRunner session, until you activate another mode.

Use WinRunner's debugging facilities when you debug a test script:

- Control the execution of your tests using the Step commands. For more information, see Chapter 29, **Debugging Test Scripts**.
- Set breakpoints to pause execution at specified points in the test script. For more information, see Chapter 30, **Using Breakpoints**.
- Use the Watch List to monitor variables in a test script during execution. For more information, see Chapter 31, **Monitoring Variables**.

Find
?

For more information about running a test script in Debug mode, see **Running a Test to Debug Your Test Script** on page 463.

Update

Use the Update mode to update the *expected* results of a test or to create a new expected results directory. For example, you could *update* the expected results for a GUI checkpoint that checks a push button, in the event that the push button default status changes from enabled to disabled. You may want to *create* an additional set of expected results if, for example, you have one set of expected results when you run your application in Windows 95 and another set of expected results when your run your application in Windows NT. For more information about generating additional sets of expected results, see **Generating Multiple Expected Results** on page 464.

Note that after a test has run in Update mode or been aborted, Verify automatically becomes the default run mode again.

By default, WinRunner saves expected results in the *exp* directory, overwriting any existing expected results.

You can update the expected results for a test in one of two ways:

- by globally overwriting the full existing set of expected results by running the entire test using a Run command

- by updating the expected results for individual checkpoints and synchronization points using the Run from Arrow command or a Step command

For more information about running a test script in Update mode, see **Running a Test to Update Expected Results** on page 464.

WinRunner Run Commands

You use the Run commands to execute your tests. When a test is running, the execution arrow in the left margin of the test script marks each TSL statement as it is interpreted.

Run from Top

The Run from Top command runs the active test from the first line in the test script. If the test calls another test, WinRunner displays the script of the called test. Execution stops at the end of the test script.

Run from Arrow

The Run from Arrow command runs the active test from the line in the script marked by the execution arrow. In all other aspects, the Run from Arrow command is the same as the Run from Top command.

Run Minimized Commands

You run a test using a Run Minimized command to make the entire screen available to the application being tested during test execution. The Run Minimized commands shrink the WinRunner window to an icon while the test runs. The WinRunner window automatically returns to its original size at the end of the test, or when you stop or pause the test run. You can use the Run Minimized commands to run a test either from the top of the test script or from the execution arrow.

Step Commands

You use a Step command to run a single statement in a test script. For more information on the Step commands, see Chapter 29, **Debugging Test Scripts**.

Stop

You can stop a test run immediately by choosing the Stop command. When you stop a test, test variables and arrays become undefined. The test options, however, retain their current values. See **Controlling Test Execution with Testing Options** on page 468 for more information.

After stopping a test, you can access only those functions that you loaded using the **load** command. You cannot access functions that you compiled using the Run commands. Recompile these functions to regain access to them. For more information, see Chapter 20, **Creating Compiled Modules**.

Pause

The Pause command pauses test execution. Unlike Stop, which immediately terminates execution, a paused test continues running until all previously interpreted TSL statements are executed. When you pause a test, test variables and arrays maintain their values, as do the test options. See **Controlling Test Execution with Testing Options** on page 468 for more information.

To resume execution of a paused test, choose the appropriate Run command. Execution resumes from the point that you paused the test.

Choosing Run Commands Using Softkeys

You can activate several of WinRunner’s commands using softkeys. WinRunner reads input from softkeys even when the WinRunner window is not the active window on your screen, or when it is minimized. Note that you can configure the default softkey configurations. For more information about configuring softkeys, see Chapter 32, **Customizing WinRunner’s User Interface**.

The following table lists the default softkey configurations for running tests:

Command	Default Softkey Combination	Function
RUN FROM ARROW	Ctrl Left + F7	Executes the test from the line in the script indicated by the arrow
RUN FROM TOP	Ctrl Left + F5	Executes the test from the beginning
STEP	F6	Executes and indicates 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 appears in the WinRunner window but is not executed.
STEP TO CURSOR	Ctrl Left + F9	Runs a test from the line executed until the line marked by the insertion point
PAUSE	PAUSE	Stops test execution after all previously interpreted TSL statements have been executed. Execution can be resumed from this point.
STOP	Ctrl Left + F3	Stops test execution

Running a Test to Check Your Application

When you run a test to check the behavior of your application, WinRunner compares the current results with the expected results. You specify the directory in which to save the verification results for the test.

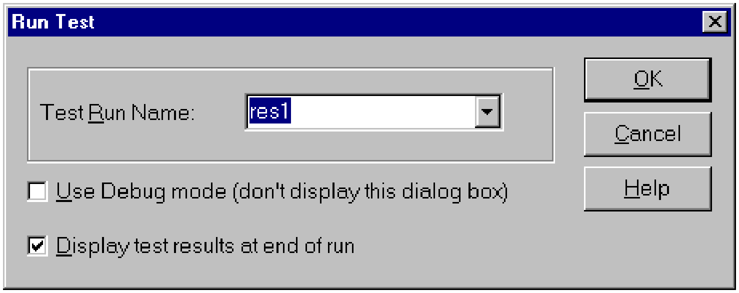
To run a test to check your application:

1 Open the test if it is not already open.

2 Make sure that Verify is selected from the drop-down list of Run modes on the Standard toolbar.

3 Choose the appropriate Run command.

The Run Test dialog box opens, displaying a default test run name for the verification results.


Run Test

Test Run Name: res1

☐ Use Debug mode (don't display this dialog box)

☑ Display test results at end of run

4 You can save the test results under the default test run name. To use a different name, type in a new name or select an existing name from the list.

To instruct WinRunner to display the test results automatically following the test run (the default), select the Display Test Results at End of Run check box.

Click OK. The Run Test dialog box closes and WinRunner runs the test according to the Run command you chose.

5 Test results are saved in the test run name you specified.

Running a Test to Debug Your Test Script

When you run a test to debug your test script, WinRunner compares the current results with the expected results. Any differences are saved in the debug results directory. Each time you run the test in Debug mode, WinRunner overwrites the previous debug results.

To run a test to debug your test script:

1. Open the test if it is not already open.
2. Select Debug from the drop-down list of run modes on the Standard toolbar.
3. Choose the appropriate Run command.

   To execute the entire test, choose Run from Top. The test runs from the top of the test script and generates a set of debug results.

   To execute part of the test, choose Run from Arrow or a Step command. The test runs according to the command you chose, and generates a set of debug results.

Find
?

Running a Test to Update Expected Results

When you run a test to update expected results, the new results replace the expected results created earlier and become the basis of comparison for subsequent test runs.

To run a test to update the expected results:

1. Open the test if it is not already open.
2. Select Update from the list of run modes on the Standard toolbar.
3. Choose the appropriate Run command.

   To update the entire set of expected results, choose Run from Top.

   To update only a portion of the expected results, choose Run from Arrow or one of the Step commands.

WinRunner runs the test according to the Run command you chose and updates the expected results. The default directory for expected results is *exp*.

Generating Multiple Expected Results

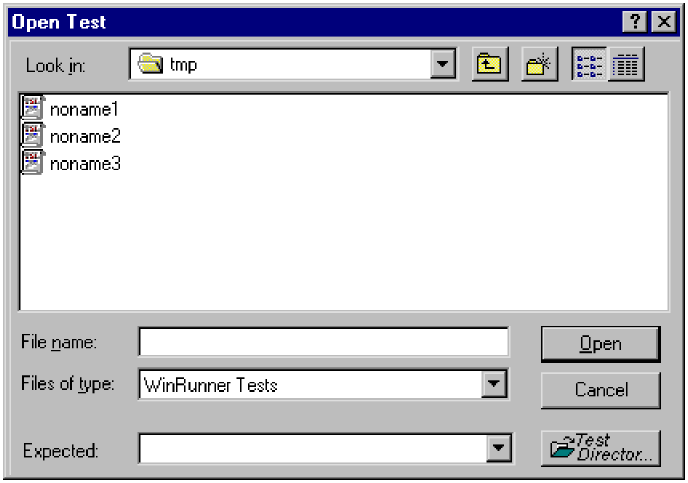
You can generate more than one set of expected results for any test. You may want to generate multiple sets of expected results if, for example, the response of your application varies according to the time of day. In such a situation, you would generate a set of expected results for each defined period of the day.

To create a different set of expected results for a test:

1 Choose Open on the File menu. The Open Test dialog box opens.

2 In the Open Test dialog box, select the test for which you want to create multiple sets of expected results. Type in a unique directory name for the new expected results in the Expected box.

Note that to create a new set of expected results for a test that is already open, choose Open on the File menu to open the Open Test dialog box, select the open test, and then enter a name for a new expected results directory in the Expected box.



3 Click OK. The Open Test dialog box closes.

4 Choose Update from the list of Run modes on the toolbar.

5 Choose Run from Top to generate a new set of expected results.

WinRunner runs the test and generates a new set of expected results, in the directory you specified.

Running a Test with Multiple Sets of Expected Results

If a test has multiple sets of expected results, you specify which expected results to use before running the test.

To run a test with multiple sets of expected results:

1 Choose Open from the File menu. The Open Test dialog box opens.

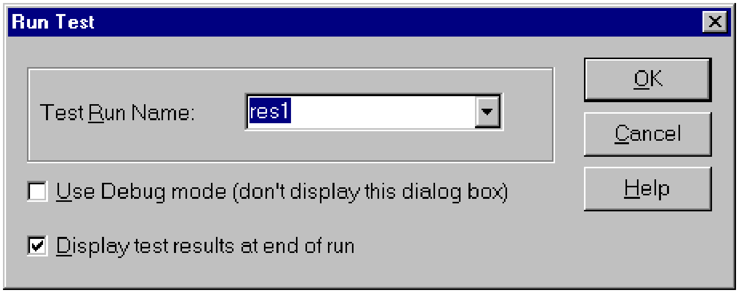
Note that if the test is already open, but it is accessing the wrong set of expected results, you must choose Open on the File menu to open the Open Test dialog box, select the open test, and then enter the correct expected results directory in the Expected box.

2 In the Open Test dialog box, click the test that you want to run. The Expected box displays all the sets of expected results for the test you chose.

3 Select the required set of expected results in the Expected box, and click OK. The Open Test dialog box closes.

4 Select Verify from the drop-down list of run modes on the Standard toolbar.

5 Choose the appropriate Run command. The Run Test dialog box opens, displaying a default test run name for the verification results—for example, *res1*.



Run Test

Test Run Name: res1

☐ Use Debug mode (don't display this dialog box)

☑ Display test results at end of run

6 Click OK to begin test execution, and to save the test results in the default directory. To use a different verification results directory, type in a new name or choose an existing name from the list.

Click OK. The Run Test dialog box closes. WinRunner runs the test according to the Run command you chose and saves the test results in the directory you specified.

Controlling Test Execution with Testing Options

You can control how a test is run using WinRunner's testing options. For example, you can set the time WinRunner waits at a bitmap checkpoint for a bitmap to appear, or the speed that a test is run.

You set testing options in the Options dialog box. Choose Options on the Settings menu to open this dialog box. You can also set testing options from within a test script using the **setvar** function.

Each testing option has a default value. For example, the default for the threshold for difference between bitmaps option (that defines the minimum number of pixels that constitute a bitmap mismatch) is 0. It can be set globally in the Run tab of the Options dialog box. For a more comprehensive discussion of setting testing options globally, see Chapter 33, **Setting Global Testing Options**.

You can also set the corresponding *min_diff* option from within a test script using the **setvar** and **getvar** functions. For a more comprehensive discussion of setting testing options from within a test script, see Chapter 34, **Setting Testing Options from a Test Script**.

If you assign a new value to a testing option, you are prompted to save this change to your WinRunner configuration when you exit WinRunner.

Find
?

You can view the settings for the current test in the Current Test tab of the Options dialog box. For more information about viewing the settings for the current test, see Chapter 33, **Setting Global Testing Options**.

Running Tests
Analyzing Test Results

After you execute a test, you can view a report of all the major events that occurred during the test run.

This chapter describes:

Find

?

About Analyzing Test Results

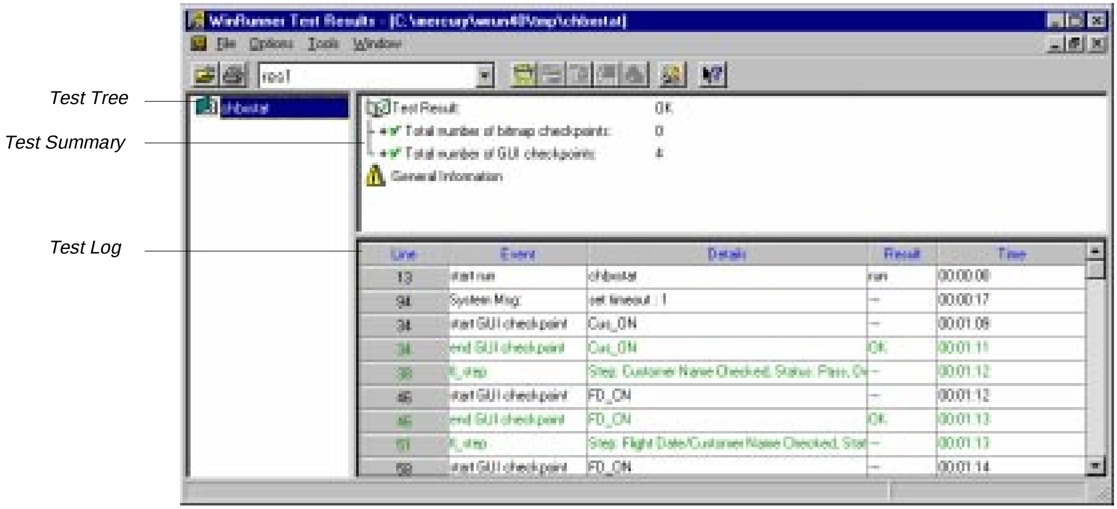
After you run a test, test results are presented in the Test Results window. This window contains a description of the major events that occurred during the test run, such as GUI and bitmap checkpoints, file comparisons, and error messages. It also includes tables and pictures to help you analyze the results.

Find

?

The Test Results Window

After a test run, you can view test results in the Test Results window. To open the window, choose Test Results on the Tools menu or click the Test Results button.


Test Tree
Test Summary
Test Log
WinRunner Test Results - [C:\mercury\wrun40\tmp\chbestat]
File Options Tools Window
res1
chbestat

Test Result: OK
Total number of bitmap checkpoints: 0
Total number of GUI checkpoints: 4
General Information

Line	Event	Details	Result	Time
13	start run	chbestat	run	00:00:08
94	System Msg	set timeout : 1	--	00:00:17
34	start GUI checkpoint	Cus_ON	--	00:01:08
34	end GUI checkpoint	Cus_ON	OK	00:01:11
38	tl_step	Step: Customer Name Checked, Status: Pass, De	--	00:01:12
46	start GUI checkpoint	FD_ON	--	00:01:12
46	end GUI checkpoint	FD_ON	OK	00:01:13
51	tl_step	Step: Flight Date/Customer Name Checked, Stat	--	00:01:13
58	start GUI checkpoint	FD_ON	--	00:01:14

Test Tree

The test tree shows all tests executed during the test run. The first test in the tree is the *calling test*. Tests below the calling test are *called tests*. To view the results of a test, click the test name in the tree.

Test Summary

The following information appears in the test summary:

Test Result

Indicates whether the test passed or failed. For a batch test, this refers to the batch test itself and not to the tests that it called. Double-click the Test Result button to view the following details:

Total number of bitmap checkpoints: The total number of bitmap checkpoints that occurred during the test run. Double-click to view a detailed list of the checkpoints. For example,

Img1 sample (5)

indicates the first bitmap checkpoint (1) in a test called *sample*, in the fifth line of the test script. The number in parentheses indicates the line in the test script that contains the **obj_check_bitmap** or **win_check_bitmap** statement.

Total number of GUI checkpoints: The total number of GUI checkpoints that occurred during the test run. Double-click to view a detailed list of the checkpoints. For example,

gui1 sample (7)

indicates the first GUI checkpoint (1) in a test called *sample*, in the seventh line of the test script. The number in parentheses indicates the line in the test script that contains the **obj_check_bitmap** or **win_check_bitmap** statement.

General Information

Double-click the General Information button to view the following test details:

Date: The date and time of the test run.

Operator Name: The name of the user who ran the test.

Expected Results Directory: The name of the expected results directory used for comparison by the GUI and bitmap checkpoints.

Total Run Time: Total time (hr:min:sec) that elapsed from start to finish of the test run.

Test Log

The test log provides detailed information on every major event that occurred during the test run. These include the start and termination of the test; GUI and bitmap checkpoints; file comparisons; changes in the progress of the test flow; changes to system variables; displayed report messages; and run time errors.

A row describing a mismatch or failure appears in red; a row describing a successful event appears in green.

Double-click the event in the log to view the following information.

- For a bitmap checkpoint, you can view the expected bitmap and the actual bitmap captured during the run. If a mismatch was detected, you can also view an image showing the differences between the expected and actual bitmaps.
- For a GUI checkpoint, you can view the results in a table. The table lists all the GUI objects included in the checkpoint and the results of the checks for each object.
- For a file comparison, you can view the two files that were compared to each other. If a mismatch was detected, the non-matching lines in the files are highlighted.

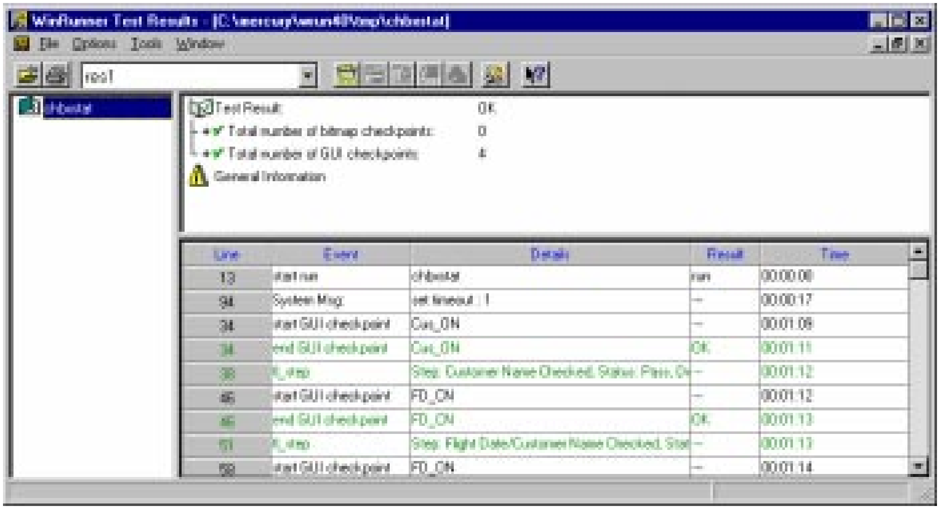
Viewing the Results of a Test Run

After a test run, you can view test results in the Test Results window. The Test Results window opens and displays the results of the current test. You can view expected, debug, and verification results in the Test Results window.

To view the results of a test run:

1 To open the Test Results window, choose Tools > Test Results, or click the Test Results button in the main WinRunner window.

Note that if you ran a test in Verify mode and the Display Test Results at End of Run option was selected (the default) in the Run Test dialog box, the Test Results window automatically opens when a test run is completed. For more information, see Chapter 25, **Running Tests**.


WinRunner Test Results - [C:\mercury\wrun4\tmp\chbestat]
File Options Tools Window
res1
chbestat

Test Result: OK
Total number of bitmap checkpoints: 0
Total number of GUI checkpoints: 4
General Information

Line	Event	Details	Result	Time
13	start run	chbestat	run	00:00:00
34	System Msg	set timeout : 1	--	00:00:17
34	start GUI checkpoint	Cus_ON	--	00:01:08
34	end GUI checkpoint	Cus_ON	OK	00:01:11
38	tl_step	Step: Customer Name Checked, Status: Pass, De	--	00:01:12
46	start GUI checkpoint	FD_ON	--	00:01:12
46	end GUI checkpoint	FD_ON	OK	00:01:13
51	tl_step	Step: Flight Date/Customer Name Checked, Stat	--	00:01:13
58	start GUI checkpoint	FD_ON	--	00:01:14

2 By default, the Test Results window displays the results of the most recently executed test run.

To view other test run results, click the results directory box and select a test run.

3 To view a text version of a report, choose Tools > Text Report from the Test Results window; the report opens in Notepad.

4 To view only specific types of results in the events column in the test log, choose Options > Filters or click the Filters button.

5 To print test results directly from the Test Results window, choose File > Print or click the Print button.

In the Print dialog box, choose the number of copies you want to print and click OK. Test results print in a text format.

6 To close the Test Results window, choose File > Exit.

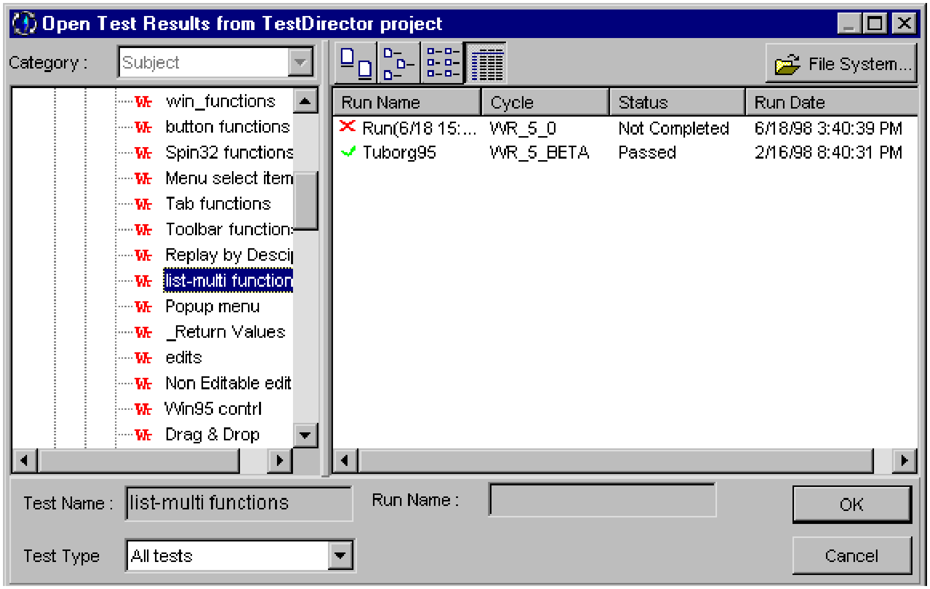
To view the results of a test run from a TestDirector database:

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 TestDirector Database dialog box opens and displays the test plan tree.


Open Test Results from TestDirector project

Category : Subject

win_functions
button functions
Spin32 functions
Menu select item
Tab functions
Toolbar function
Replay by Desci
list-multi function
Popup menu
_Return Values
edits
Non Editable edit
Win95 contrl
Drag & Drop

Run Name	Cycle	Status	Run Date
Run(6/18 15:...	WR_5_0	Not Completed	6/18/98 3:40:39 PM
Tuborg95	WR_5_BETA	Passed	2/16/98 8:40:31 PM

Test Name : list-multi functions

Run Name :

Test Type : All tests

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. The *Run Name* column displays whether your test run passed or failed and contains the names of the test runs. The *Cycle* 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.

See the previous section for an explanation of the options in the Test Results window.

Note: For more information on viewing the results of a test run from a TestDirector database, see Chapter 37, **Managing the Testing Process**.

Find
?

Viewing the Results of a GUI Checkpoint

A GUI checkpoint helps you to identify changes in the look and behavior of GUI objects in your application. The results of a GUI checkpoint are presented in the GUI Checkpoint Results dialog box that you open from the Test Results window. It lists each object included in the GUI checkpoint and the type of checks performed. Each check is listed as either passed or failed, and the expected and actual results are shown. If one or more objects fail, the entire GUI checkpoint is marked as failed in the test log.

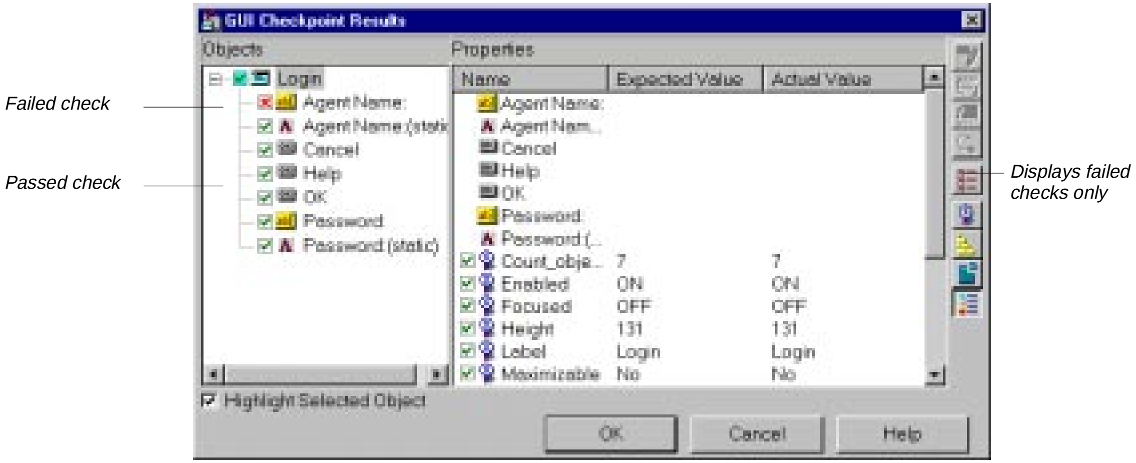
For more information, see Chapter 9, **Checking GUI Objects**.

To display the results of a GUI checkpoint:

1 Open the Test Results window. In the test log, look for entries that list “end GUI checkpoint” in the Event column. Failed GUI checkpoints appear in red; passed GUI checkpoints appear in green.

2 Double-click an “end GUI checkpoint” entry in the test log or click the entry and click Display. The GUI Checkpoint Results dialog box opens.

The GUI Checkpoint Results dialog box lists the results of the selected checkpoint.



Objects:

- Login
  - Agent Name:
  - Agent Name (static
  - Cancel
  - Help
  - OK
  - Password
  - Password (static)

Name	Expected Value	Actual Value
Agent Name:		
Agent Nam...		
Cancel		
Help		
OK		
Password		
Password (...		
Count_obje...	7	7
Enabled	ON	ON
Focused	OFF	OFF
Height	131	131
Label	Login	Login
Maximizable	No	No

☑ Highlight Selected Object

The dialog box lists every object checked and the types of checks performed. Each check is marked as either passed or failed and the expected and the actual results are shown.

3 Click OK to close the dialog box.

Viewing the Results of a Properties Check on ActiveX Controls

You can view the results of a properties check of an ActiveX control class in the Properties Differences Display window. For more information on properties check, see Chapter 10, **Working with ActiveX Controls**.

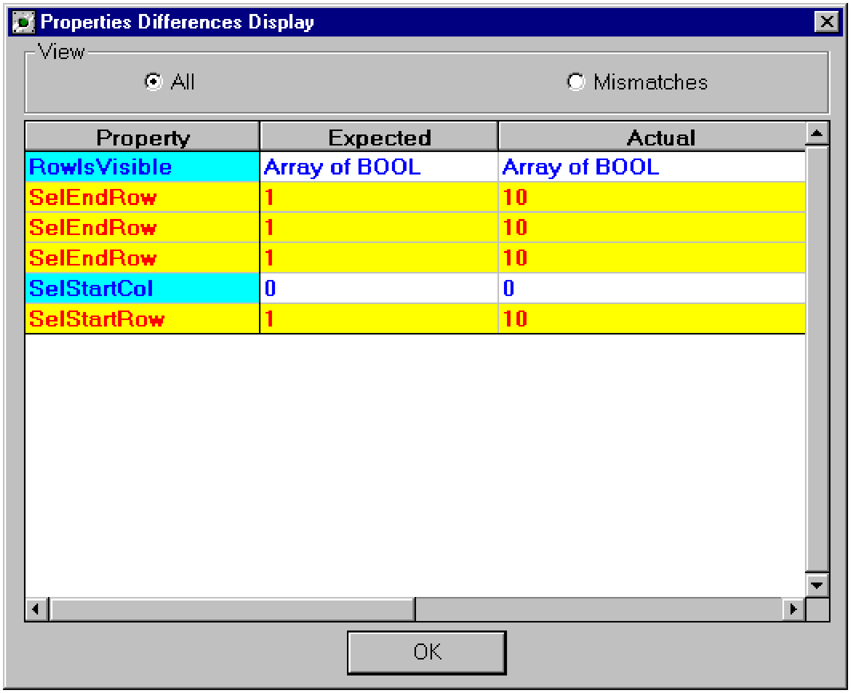
To display the results of a GUI checkpoint on ActiveX controls:

1. To open the Test Results window, choose Tools > Test Results, or click the Test Results button in the main WinRunner window.
2. Double-click an “end GUI checkpoint statement” in the test log or click the entry and click Display. The GUI Checkpoint Results dialog box opens and the results of the selected GUI checkpoint appear.

Find

?

3 Click the name of a check and click Display, or double-click the name. The Properties Differences Display window opens.



View: [✓ All] Mismatches

Property	Expected	Actual
RowIsVisible	Array of BOOL	Array of BOOL
SelEndRow	1	10
SelEndRow	1	10
SelEndRow	1	10
SelStartCol	0	0
SelStartRow	1	10

The Properties Differences Display window shows the results of a properties GUI checkpoint for a table or an ActiveX control. In the case of a mismatch, it displays both expected and actual results.

Cells in the Properties Differences Display window are color coded.

- **Blue on white background:** No mismatch
- **Red on yellow background:** Mismatch

4 To view only properties with a mismatch, click Mismatches at the top of the window.

5 Click OK to close the Properties Differences Display window.

Viewing the Results of a Contents Check on a Table

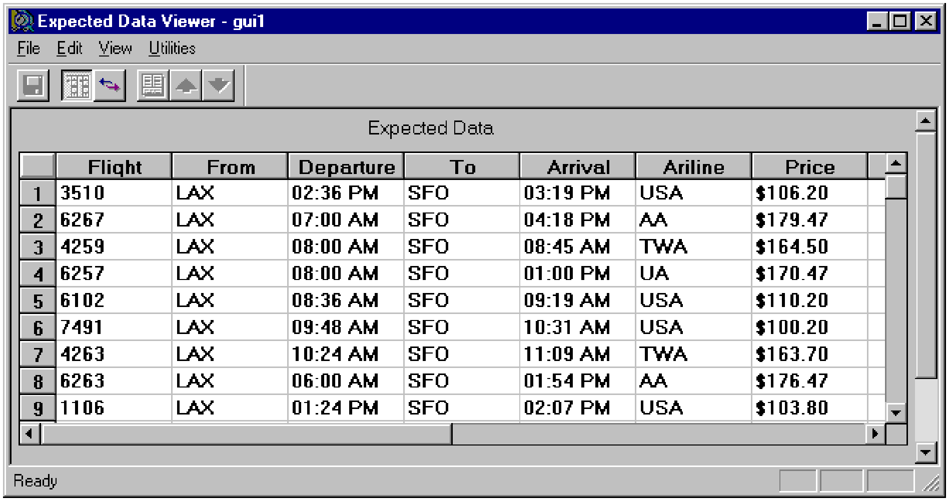
You can view the results of a contents check on a table in the Data Comparison Viewer and Expected Data Viewer. For a test run with a mismatch, the Data Comparison Viewer opens. It shows both expected and actual results. When you have a run with no mismatch, the Expected Data Viewer opens. It shows expected results only. For more information on checking the contents of a table, see Chapter 11, **Checking Tables**.

To display the results of a GUI checkpoint on table contents:

1. To open the Test Results window, choose Tools > Test Results, or click the Test Results button in the main WinRunner window.
2. Double-click an “end GUI checkpoint statement” in the test log or click the entry and click Display. The GUI Checkpoint Results dialog box opens and the results of the selected GUI checkpoint appear.
3. Click the name of a check and click Display, or double-click the name.

   For a test run with no mismatch, the Expected Data Viewer opens. When you have a run with a mismatch the Data Comparison Viewer opens.

4 If the Expected Data Viewer opens, it shows expected results only. To close the window, choose File > Exit.



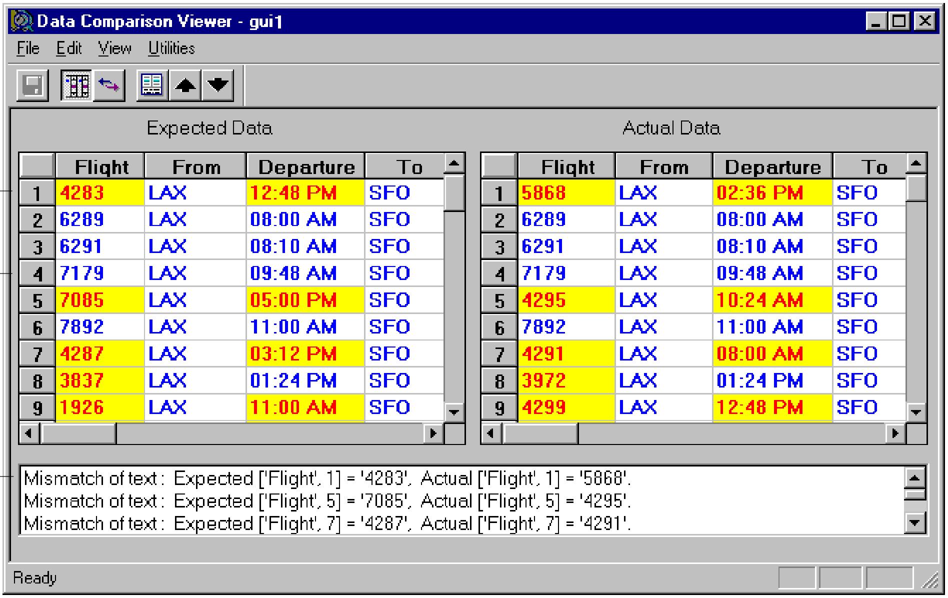
Expected Data

	Flight	From	Departure	To	Arrival	Ariline	Price
1	3510	LAX	02:36 PM	SFO	03:19 PM	USA	$106.20
2	6267	LAX	07:00 AM	SFO	04:18 PM	AA	$179.47
3	4259	LAX	08:00 AM	SFO	08:45 AM	TWA	$164.50
4	6257	LAX	08:00 AM	SFO	01:00 PM	UA	$170.47
5	6102	LAX	08:36 AM	SFO	09:19 AM	USA	$110.20
6	7491	LAX	09:48 AM	SFO	10:31 AM	USA	$100.20
7	4263	LAX	10:24 AM	SFO	11:09 AM	TWA	$163.70
8	6263	LAX	06:00 AM	SFO	01:54 PM	AA	$176.47
9	1106	LAX	01:24 PM	SFO	02:07 PM	USA	$103.80

Find

?

5 If the Data Comparison Viewer opens, it shows both expected and actual results. All cells are color coded, and all errors and mismatches are listed at the bottom of the window.


Cell contains a mismatch
Cell does not contain a mismatch
List of errors and mismatches
Data Comparison Viewer - gui1
File Edit View Utilities
Expected Data
Actual Data
Ready

Expected Data

	Flight	From	Departure	To
1	4283	LAX	12:48 PM	SFO
2	6289	LAX	08:00 AM	SFO
3	6291	LAX	08:10 AM	SFO
4	7179	LAX	09:48 AM	SFO
5	7085	LAX	05:00 PM	SFO
6	7892	LAX	11:00 AM	SFO
7	4287	LAX	03:12 PM	SFO
8	3837	LAX	01:24 PM	SFO
9	1926	LAX	11:00 AM	SFO

Actual Data

	Flight	From	Departure	To
1	5868	LAX	02:36 PM	SFO
2	6289	LAX	08:00 AM	SFO
3	6291	LAX	08:10 AM	SFO
4	7179	LAX	09:48 AM	SFO
5	4295	LAX	10:24 AM	SFO
6	7892	LAX	11:00 AM	SFO
7	4291	LAX	08:00 AM	SFO
8	3972	LAX	01:24 PM	SFO
9	4299	LAX	12:48 PM	SFO

```
Mismatch of text :  Expected ['Flight', 1] = '4283',  Actual ['Flight', 1] = '5868'.
Mismatch of text :  Expected ['Flight', 5] = '7085',  Actual ['Flight', 5] = '4295'.
Mismatch of text :  Expected ['Flight', 7] = '4287',  Actual ['Flight', 7] = '4291'.
```

Use the following color codes to interpret the differences that are highlighted in your window:

- **Blue on white background:** Cell was included in the comparison and no mismatch was found.
- **Cyan on ivory background:** Cell was not included in the comparison.
- **Red on yellow background:** Cell contains a mismatch.
- **Magenta on green background:** Cell was verified but not found in the corresponding table.
- **Background color only:** cell is empty (no text).

6 By default, scrolling between the Expected and Actual tables in the Data Comparison Viewer is synchronized. When you click on any cell, the corresponding cell in the other table flashes red.

To scroll through the tables separately, clear Synchronize Scrolling from the Utilities menu. Use the scroll bar as needed to view hidden parts of the table.

7 To filter a list of errors and mismatches that appear at the bottom of the Data Comparison Viewer, use the following options:

- **To view mismatches for a specific column only:** Double-click a column heading (the column name) in either table.
- **To view mismatches for a single row:** Double-click a row number in either table.
- **To view mismatches for a single cell:** Double-click a cell with a mismatch.
- **To see all mismatches:** Choose Utilities > List All Mismatches or double-click the empty cell in the upper left corner of the table.

- **To clear the list:** Double-click a cell with no mismatch.
- **To see the cell(s) that correspond to a listed mismatch:** Click a mismatch in the list at the bottom of the dialog box to see the corresponding cells in the table flash red. If the cell with the mismatch is not visible, one or both table scroll automatically to display it.

8 Choose File > Exit to close the Data Comparison Viewer.

Find
?

Editing the Expected Results of Contents Check on a Table

You can edit your expected results table in the Data Comparison Viewer and the Expected Data Viewer.

To edit the expected results for a GUI checkpoint on contents of a table:

1 Right-click a cell in the Expected Data table.

2 Type a new value in the cell, or use the Insert key to edit the contents.

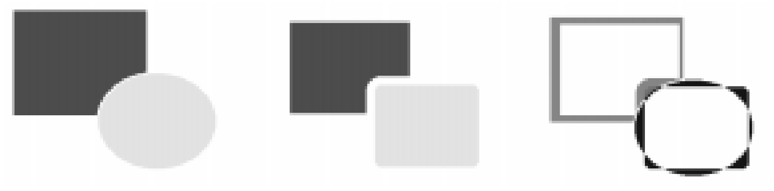
3 Click the Save Expected Data button. The expected results are updated to reflect your changes.

Viewing the Results of a Bitmap Checkpoint

A bitmap checkpoint compares expected and actual bitmaps in your application. In the Test Results window you can view pictures of the expected and actual results. If a mismatch is detected by a bitmap checkpoint during a verification run, you can also view an image showing the differences between the expected and the actual results.

To view the results of a bitmap checkpoint:

1. In the test log, look for entries that list bitmap checkpoints in the Event column.
2. To view the results of a specific bitmap checkpoint, double-click its entry in the log or click the entry and then click Display. For a verification mismatch, the expected, actual, and difference bitmaps are displayed; for all other runs, only the expected bitmaps are displayed.



Expected *Actual* *Difference*

Note: You can control which types of bitmaps are displayed (expected, actual, difference) when you view the results of a bitmap checkpoint. To set the controls, choose Options > Bitmap Controls in the Test Results window.

3 To remove a bitmap from the screen, double-click the system menu button in the bitmap window.

Find

?

Updating the Expected Results of a Checkpoint

If a bitmap or GUI checkpoint fails, you can update the expected results directory (*exp*) with data in the verification results directory. The next time you run the test, the new expected results will be compared to the current results in the application.

To update the expected results of a bitmap checkpoint:

1. Click a mismatched bitmap checkpoint in the test log.
2. Choose Update on the Options menu or click the Update button.
3. A dialog box warns that overwriting expected results cannot be undone. Click Yes to update the results.

To update the expected results of a GUI checkpoint:

1. In the test log, click a mismatched GUI checkpoint.
2. To update the results for the entire GUI checkpoint, choose Update on the Options menu or click the Update button.
3. To update the results for a specific check within the GUI checkpoint, double-click the GUI checkpoint entry in the log or click the Display button. The GUI Checkpoint Results dialog box opens. Click a failed check and click the Update Selected Check button.

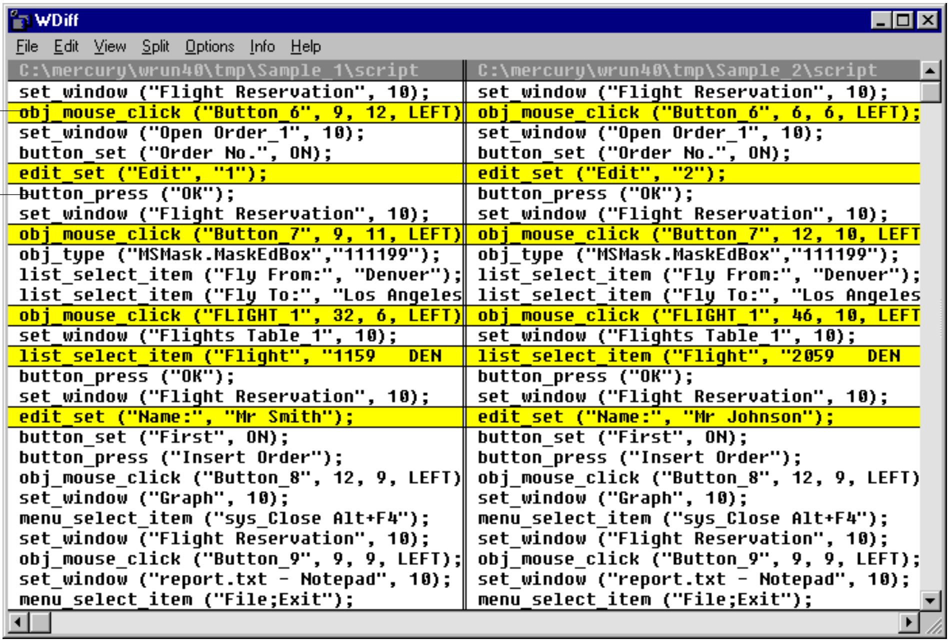
Viewing the Results of a File Comparison

If you used a **file_compare** statement in a test script to compare the contents of two files, you can view the results using the WDiff utility. This utility is accessed from the Test Results window.

To view the results of a file comparison:

1 Open the Test Results window.

2 Double-click a “file compare” event in the test log, or click the event and click Display. The WDiff utility window opens.


Line contains a mismatch
Line does not contain a mismatch
WDiff
File Edit View Split Options Info Help

```
C:\mercury\wrun40\tmp\Sample_1\script
set_window ("Flight Reservation", 10);
obj_mouse_click ("Button_6", 9, 12, LEFT)
set_window ("Open Order_1", 10);
button_set ("Order No.", ON);
edit_set ("Edit", "1");
button_press ("OK");
set_window ("Flight Reservation", 10);
obj_mouse_click ("Button_7", 9, 11, LEFT)
obj_type ("MSMask.MaskEdBox","111199");
list_select_item ("Fly From:", "Denver");
list_select_item ("Fly To:", "Los Angeles
obj_mouse_click ("FLIGHT_1", 32, 6, LEFT)
set_window ("Flights Table_1", 10);
list_select_item ("Flight", "1159   DEN
button_press ("OK");
set_window ("Flight Reservation", 10);
edit_set ("Name:", "Mr Smith");
button_set ("First", ON);
button_press ("Insert Order");
obj_mouse_click ("Button_8", 12, 9, LEFT)
set_window ("Graph", 10);
menu_select_item ("sys_Close Alt+F4");
set_window ("Flight Reservation", 10);
obj_mouse_click ("Button_9", 9, 9, LEFT);
set_window ("report.txt - Notepad", 10);
menu_select_item ("File;Exit");
```

```
C:\mercury\wrun40\tmp\Sample_2\script
set_window ("Flight Reservation", 10);
obj_mouse_click ("Button_6", 6, 6, LEFT);
set_window ("Open Order_1", 10);
button_set ("Order No.", ON);
edit_set ("Edit", "2");
button_press ("OK");
set_window ("Flight Reservation", 10);
obj_mouse_click ("Button_7", 12, 10, LEFT
obj_type ("MSMask.MaskEdBox","111199");
list_select_item ("Fly From:", "Denver");
list_select_item ("Fly To:", "Los Angeles
obj_mouse_click ("FLIGHT_1", 46, 10, LEFT
set_window ("Flights Table_1", 10);
list_select_item ("Flight", "2059   DEN
button_press ("OK");
set_window ("Flight Reservation", 10);
edit_set ("Name:", "Mr Johnson");
button_set ("First", ON);
button_press ("Insert Order");
obj_mouse_click ("Button_8", 12, 9, LEFT)
set_window ("Graph", 10);
menu_select_item ("sys_Close Alt+F4");
set_window ("Flight Reservation", 10);
obj_mouse_click ("Button_9", 9, 9, LEFT);
set_window ("report.txt - Notepad", 10);
menu_select_item ("File;Exit");
```

The WDiff utility displays both files. Lines in the file that contain a mismatch are highlighted. The file defined in the first parameter of the **file_compare** statement is on the left side of the window.

To see the next mismatch in a file, choose Next Diff on the View menu or click Tab. The window scrolls to the next highlighted line. To see the previous difference, choose Prev Diff or press the backspace key.

You can choose to view only the lines in the files that contain a mismatch. To filter file comparison results, choose Hide Matching Areas on the Options > View menu. The window shows only the highlighted parts of both files.

3 Choose File > Exit to close the WDiff Utility.

Reporting Defects Detected during a Test Run

If a test run detects a defect in the application under test, you can report it directly from the Test Results window. To report a defect, click the Report Defects button or choose Report Bug in the Tools menu. The Remote Defect Reporter dialog box opens so that you can type details of the defect. You can then send this information to a file or send it by e-mail. The information on the defect can later be imported into TestDirector where a quality assurance manager determines its severity and assigns it to a developer to be fixed. See Chapter 39, **Reporting Defects** for more information.

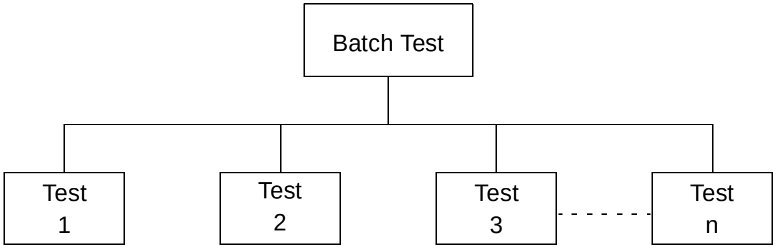
Running Batch Tests

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

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 tab of the Options dialog box before you execute the test.

When you run a test in Batch mode, WinRunner suppresses all messages that would ordinarily be displayed during execution, such as a message reporting a bitmap mismatch. WinRunner also suppresses all **pause** statements and any halts in execution 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.

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 execute a group of tests from the command line. For details, see Chapter 28, **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 Batch Run in Batch Mode option in the Run tab of the 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 16, **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\\save" ();
    call "c:\\pbtests\\setup" ();
    }
```

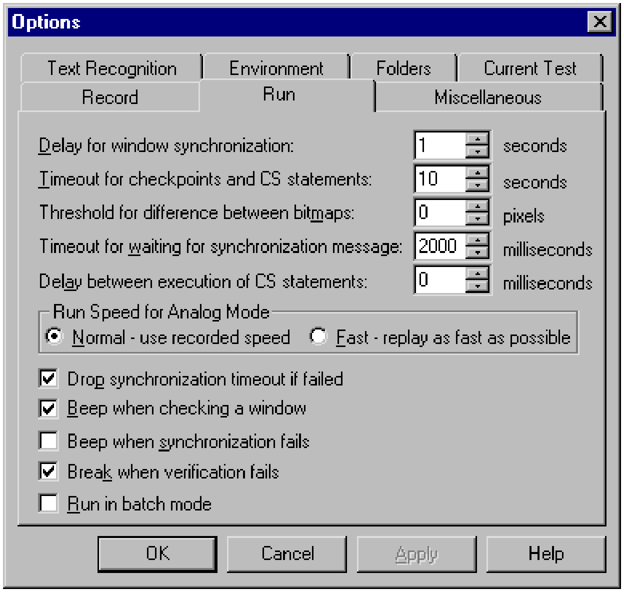
To enable a batch test:

1 Choose Options on the Settings menu.

The Options dialog box opens.

2 Click the Run tab.

3 Select the Run in Batch Mode check box.



Delay for window synchronization: 1 seconds
Timeout for checkpoints and CS statements: 10 seconds
Threshold for difference between bitmaps: 0 pixels
Timeout for waiting for synchronization message: 2000 milliseconds
Delay between execution of CS statements: 0 milliseconds

Run Speed for Analog Mode
[✓ Normal - use recorded speed] Fast - replay as fast as possible

- [x] Drop synchronization timeout if failed
- [x] Beep when checking a window
- [ ] Beep when synchronization fails
- [x] Break when verification fails
- [ ] Run in batch mode

4 Click OK to close the Options dialog box.

For more information on setting the batch option in the Options dialog box, see Chapter 33, **Setting Global Testing Options**.

Running a Batch Test

You execute a batch test in the same way that you execute a regular test. Choose a mode (Verify, Update, or Debug) from the list on the toolbar and choose Run from Top on the Run menu. See Chapter 25, **Running Tests**, for more information.

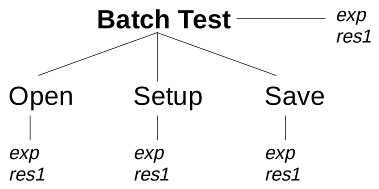
When you run a batch test, WinRunner opens and executes each called test. All messages are suppressed so that the tests are run without interruption. If you run the batch test in Verify mode, the current test results are compared to the expected test results saved earlier. If you are running the batch test in order to update expected results, new expected results are created in the expected results directory for each test. See **Storing Batch Test Results** on page 504 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 subdirectory under the test. The same is true when a test is called by a batch test. WinRunner saves the results for each called test separately in a subdirectory under the test. A subdirectory is also created for the batch test that contains the overall results of the batch test run.

For example, suppose you create three tests, *Open*, *Setup*, and *Save.* For each test, expected results are saved in a *exp* subdirectory under the test directory. Suppose you also create a batch test that calls the three tests. Prior to executing the batch test in Verify mode, you tell WinRunner to save the results in a directory called *res1*. When the batch test is run, it compares the current test results to the expected results saved earlier. Under each test directory, WinRunner creates a subdirectory called *res1* in which it saves the verification results for the test. A *res1* directory 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* subdirectory for each test and for the batch test.

Note that if you run the batch test without selecting the Run in Batch Mode check box in the Options dialog box, WinRunner saves results only in a subdirectory for the batch test. This can cause problems at a later stage if you choose to run the tests separately, since WinRunner will not know where to look for the previously saved expected and verification results.

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, then the batch test is marked as failed.

The test log section of 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 in the log. To view the results of the called test, double-click its *call_test* entry. For more information on viewing test results in the Test Results window, see Chapter 26, **Analyzing Test Results**.

Running Tests from the Command Line

You can run tests directly from the Windows command line.

This chapter describes:

- 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 parameters. 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.

Using the command line, you can:

- 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 execution parameters 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:

```
C\WRUN.EXE -t c:\batch\newclock -batch on -run_minimized -dont_quit
```

The test *newclock* is loaded and then executed in batch mode with WinRunner minimized. WinRunner remains open after the test run is completed.

Using the Windows Command Line

You can use the Windows command line to start WinRunner with predefined parameters. If you plan to use the same set of parameters 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 Choose Run on the Start menu. 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.

Adding a Custom WinRunner Shortcut

You can make the parameters 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.

Find
?

Command Line Options

Following is a description of each command line option.

-animate

Instructs WinRunner to execute the loaded test, while the execution arrow displays the line of the test being run.

-auto_load {on | off}

Activates or deactivates automatic loading of the temporary GUI map file.
(Default = **on**)

-auto_load_dir pathname

Determines the directory 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**)

-beep {on | off}

Activates or deactivates the WinRunner system beep.
(Default = **on**)

-create_text_report {on | off}

Instructs WinRunner to write test results to a text report, called *report.txt*, which is saved in the results directory.
(Default = **off**)

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])

-delay *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 seconds) that WinRunner waits between consecutive samplings of the screen. If two consecutive checks produce the same results, WinRunner captures the window or object.
(Default = **1** [second])

-dont_quit

Instructs WinRunner not to close after completing the test.

-dont_show_welcome

Instructs WinRunner not to show the Welcome window.

Find
?

-exp *expected_results_name*

Designates a name for the subdirectory 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**)

-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 that if a command line option appears both in the command line and in the file, WinRunner uses the option in the file.

-fontgrp *group_name*

Specifies the active font group when WinRunner is started.

-ini *initialization_test_name*

Defines the .ini file that is used when WinRunner is started.

-min_diff *non-negative integer*

Defines the number of pixels that constitute the threshold for an image mismatch.
(Default = **0** [pixels])

Find
?

-mismatch_break {on | off}

Activates or deactivates Break on Mismatch before a verification run. The functionality of Break on Mismatch 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 test execution.
(Default = **off**)

-run_minimized

Instructs WinRunner to run tests with WinRunner minimized to an icon. Note that specifying this option does not execute the tests.

-search_path *pathname*

Defines the directories to be searched for tests to be opened and/or called. The search path is given as a string.
(Default = **startup directory** and **installation directory/lib**)

-speed {normal | fast}

Sets the speed for the execution of the loaded test.
(Default = **fast**)

-t *testname*

Specifies the name of the test to be loaded in the WinRunner window. This can be the name of a test stored in a directory specified in the search path or the full pathname of any test stored in your system.

-td_connection {on | off}

Activates or deactivates WinRunner's connection to TestDirector.

-td_cycle_name *cycle_name*

Specifies the name of the current test cycle. This option is applicable only when WinRunner is connected to TestDirector.

-td_database_name *database_pathname*

Specifies the active TestDirector database. WinRunner can open, execute, and save tests in this database. This option is applicable only when WinRunner is connected to TestDirector.

-td_logname_dir

Defines the full pathname for an event log file. Note that this file is not a TestDirector file.

-td_password

Specifies the password for connecting to a database in a TestDirector server.

-td_server_name

Specifies the name of the TestDirector server to which WinRunner connects.

-td_user_name *user_name*

Specifies the name of the user who is currently executing a test cycle. (Formerly *user*.)

-timeout *non-negative integer*

Defines the global *timeout* variable (in seconds) used by WinRunner.
(Default = **1** [second])

-tslinit_exp

Directs WinRunner to the expected directory to be used when the *tslinit* script is running.

-verify *verification_results_name*

Specifies that the test is to be run in Verify mode and designates the name of the subdirectory in which the test results are stored.

Debugging Tests

Debugging Test Scripts

Controlling test execution can help you to identify and eliminate defects in your test scripts.

This chapter describes:

- **Running a Single Line of a Test Script**
- **Running a Section of a Test Script**
- **Pausing Test Execution**

About Debugging Test Scripts

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 test execution.

You can also control test execution 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. For more information, see Chapter 30, **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. For more information, see Chapter 31, **Monitoring Variables**.

When you debug a test script, you run the test in the Debug mode. The results of the test are saved in a *debug* directory. Each time you run the test, the previous debug results are overwritten. Continue to run the test in the Debug mode until you are ready to run it in Verify mode. For more information on using the Debug mode, see Chapter 25, **Running Tests**.

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

The Step command executes 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.

Step Into

The Step Into command executes 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:

- The test script of the called test or function is displayed in the WinRunner window.
- The called test or function is not executed. Use Step or Step Out to continue test execution.

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 execution.

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 Step to Cursor on the Run menu or press the corresponding softkey. WinRunner executes the test up to the line marked by the insertion point.

Pausing Test Execution

You can temporarily suspend test execution by choosing the Pause command or by adding a **pause** statement to your test script.

Pause Command

You can suspend the execution of a test by clicking Pause in the Run menu 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 execution of a paused test, click the appropriate Run command in the Run 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 test execution and displays the time that elapsed between two points.

```
t1=get_time();
t2=get_time();
pause ("Time elapsed" is & t2-t1);
```

For more information on the **pause** function, refer to the *TSL Online Reference*.

Find
?

Debugging Tests
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:

Find

?

About 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 test execution when it reaches a breakpoint. You can examine the effects of the test run up to the breakpoint, make any necessary changes, and then restart the test from the breakpoint. Use the Run from Arrow command to restart the test run. Once restarted, the test continues until the next breakpoint is encountered or the test is completed.

You can use breakpoints to:

- suspend test execution and inspect the state of your application
- monitor the entries in the Watch List. See Chapter 31, **Monitoring Variables**, for more information.
- mark a point from which to begin stepping through a test script using the Step commands. See Chapter 29, **Debugging Test Scripts**, 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 have to redefine breakpoints to continue debugging the script in another session.

Breakpoint Types

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 Dialog box 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 the test will stop every time the breakpoint is passed.

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 dialog box as:

```
ui_func [ui_test : 25] : 10
```

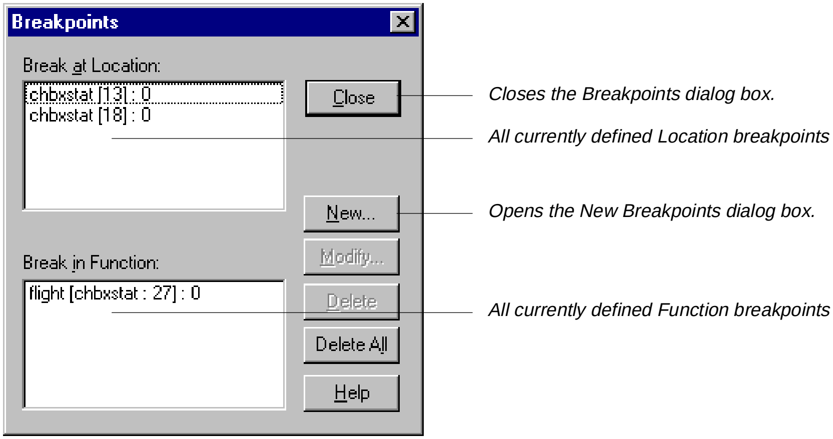
This indicates that a breakpoint has been defined for the line containing the function ui_func, in the compiled module ui_test: in this case line 25. The pass count is set to 10, meaning that the test will be stopped each time the function has been called ten times.

Setting Break at Location Breakpoints

You set Break at Location breakpoints using the Breakpoints dialog box, the mouse, or the Toggle Breakpoint command.

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

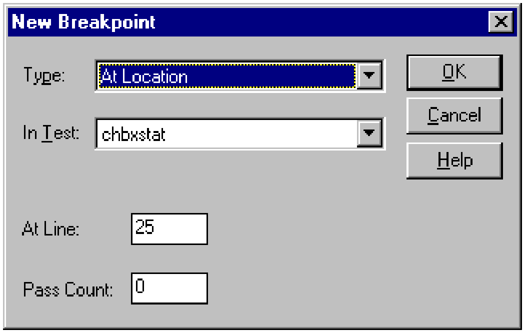
1 Choose Breakpoints on the Debug menu to open the Breakpoints dialog box.


Breakpoints
Break at Location:
Close
New...
Modify...
Delete
Delete All
Help
Break in Function:
Closes the Breakpoints dialog box.
All currently defined Location breakpoints
Opens the New Breakpoints dialog box.
All currently defined Function breakpoints

chbxstat [13] : 0
chbxstat [18] : 0

flight [chbxstat : 27] : 0

2 Click New to open the New Breakpoint dialog box.



New Breakpoint

Type: At Location

In Test: chbxstat

At Line: 25

Pass Count: 0

3 Select the breakpoint type and the test name. Modify the line number and the pass count as required.

4 Click OK to set the breakpoint and close the New Breakpoint dialog box. The new breakpoint appears in the Break at Location list in the Breakpoints dialog box.

5 Click Close to close the Breakpoints dialog box.

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 Move the execution arrow to the line in the test script at which you want test execution to stop. To move the arrow, click inside the margin next to the desired line in the test script.

2 Click the right mouse button. The breakpoint symbol (!) appears in the left margin of the WinRunner window.

To remove the breakpoint, click the breakpoint symbol with the right mouse button or choose the Toggle Breakpoint command on the Debug menu.

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 Toggle Breakpoint on the Debug menu. The breakpoint symbol (!) appears in the left margin of the WinRunner window.

To remove the breakpoint, click the breakpoint symbol with the right mouse button or choose the Toggle Breakpoint command on the Debug menu.

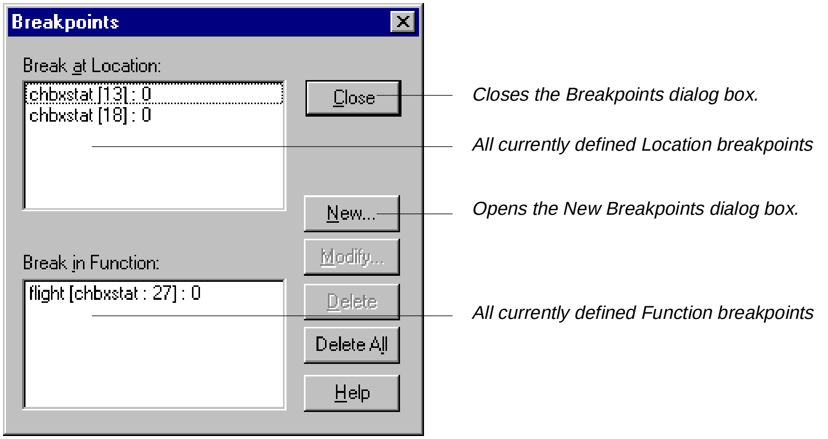
Find
?

Setting Break in Function Breakpoints

A Break in Function breakpoint stops test execution at the user-defined function that you specify. You can set a Break in Function breakpoint using either the Breakpoints dialog box or the Break in Function command.

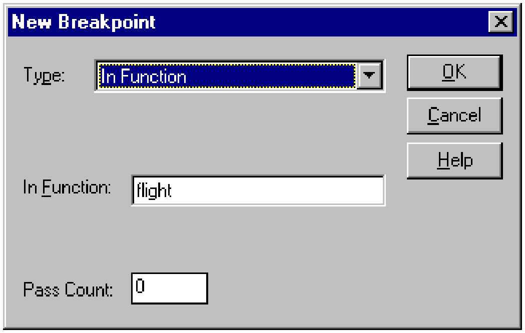
To set a Break in Function breakpoint using the Breakpoints dialog box:

1 Choose Breakpoints on the Debug menu to open the Breakpoints dialog box.


Breakpoints
Break at Location:
chbxstat [13] : 0
chbxstat [18] : 0
Break in Function:
flight [chbxstat : 27] : 0
Close
New...
Modify...
Delete
Delete All
Help
Closes the Breakpoints dialog box.
All currently defined Location breakpoints
Opens the New Breakpoints dialog box.
All currently defined Function breakpoints

2 Click New to open the New Breakpoint dialog box.

3 In the Type box, select In Function. The dialog box changes so that you can type in a function name and a pass count value.


New Breakpoint

Type: In Function

In Function: flight

Pass Count: 0

4 Type the name of a user-defined function in the In Function box. The function must be compiled by WinRunner. For more information, see Chapter 19, **Creating User-Defined Functions**, and Chapter 20, **Creating Compiled Modules**.

5 Type a value in the Pass Count box.

6 Click OK to set the breakpoint and close the New Breakpoint dialog box.

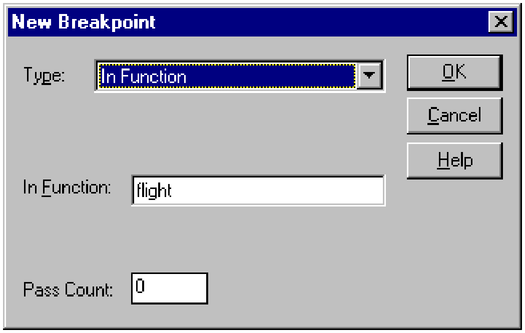
The new breakpoint appears in the Break in Function list of the Breakpoints dialog box.

7 Click Close to close the Breakpoints dialog box.

The breakpoint symbol (!) appears in the left margin of the WinRunner window.

To set a Break in Function breakpoint using the Break in Function command:

1 Choose Break in Function on the Debug menu. The New Breakpoint dialog box opens.



Type: In Function
In Function: flight
Pass Count: 0

2 Type the name of a user-defined function in the In Function box. The function must be compiled by WinRunner. For more information, see Chapter 19, **Creating User-Defined Functions**, and Chapter 20, **Creating Compiled Modules**.

3 Type a value in the Pass Count box.

4 Click OK. The breakpoint symbol (!) appears in the left margin of the WinRunner window.

5 Click Close to set the breakpoint and close the New Breakpoint dialog box.

Modifying Breakpoints

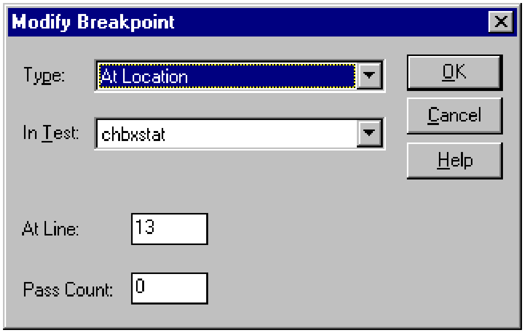
You can modify the definition of a breakpoint using the Breakpoints dialog box. You can change the breakpoint's type, the test or line number for which it is defined, and the value of the pass count.

To modify a breakpoint:

1 Choose Breakpoints on the Debug menu to open the Breakpoints dialog box.

2 Select a breakpoint in the Break at Location or the Break in Function list.

3 Click Modify to open the Modify Breakpoint dialog box.



Type: At Location
In Test: chbxstat
At Line: 13
Pass Count: 0

4 To change the type of Breakpoint, select a different breakpoint type in the Type box.

To select another test, select its name in the In Test box.

To change the line number at which the breakpoint will appear, type a new value in the At Line box.

To change the pass count, type a new value in the Pass Count box.

5 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 Choose Breakpoints on the Debug menu to open the Breakpoints dialog box.

2 Select a breakpoint in either the Break at Location or the Break in Function list.

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

4 Click Close to close the Breakpoints dialog box.

Note that the breakpoint symbol is removed from the left margin of the WinRunner window.

To delete all breakpoints:

1 Open the Breakpoints dialog box.

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

3 Click Close to close the dialog box.

Note that all breakpoint symbols are removed from the left margin of the WinRunner window.

Debugging Tests
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.

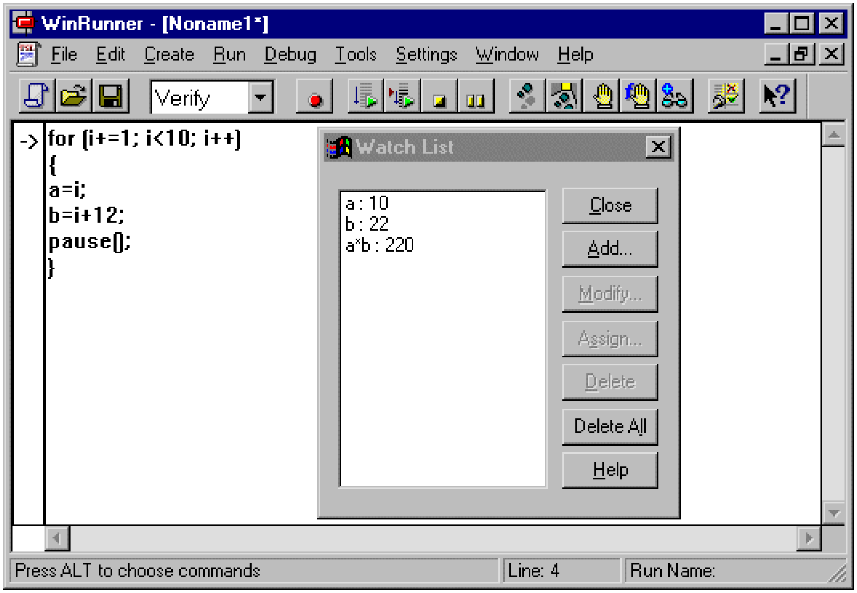
This chapter describes:

Find

?

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 simply add the elements that you want to monitor to the Watch List. During a test run, you can view the current values at each break in execution—such as after a Step command, at a breakpoint, or at the end of a test.



```
-> for (i+=1; i<10; i++)
   {
   a=i;
   b=i+12;
   pause();
   }
```

```
a : 10
b : 22
a*b : 220
```

Press ALT to choose commands | Line: 4 | Run Name:

For example, in the following test, the Watch List is used to measure and track the values of variables *a* and *b*. After each loop is executed, the test pauses so that you can view the current values.

```
for (i = 1; i <= 10; i++)
{
a = i;
b = i+12;
pause ();
}
```

After WinRunner executes the first loop, the test pauses. The variables and their values are displayed in the Watch List as follows:

```
a:1
b:13
```

When WinRunner completes the test run, the Watch List shows the following results:

```
a:10
b:22
```

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 variable for the test that WinRunner is interpreting.

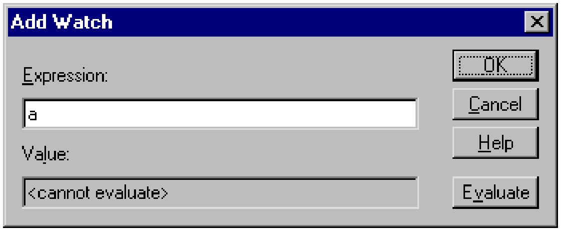
If you choose a test in the Calls list (Debug > Calls), the context of the variables and expressions in the Watch List changes. WinRunner automatically updates their values in the Watch List.

Adding Variables to the Watch List

You add variables, expressions, and arrays to the Watch List using the Add Watch dialog box.

To add a variable, an expression, or an array to the Watch List:

1 Choose Add Watch on the Debug menu to open the Add Watch dialog box.



Note that you can also open the Add Watch dialog box from the Watch List. Choose Watch List on the Debug menu and click Add.

2 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.

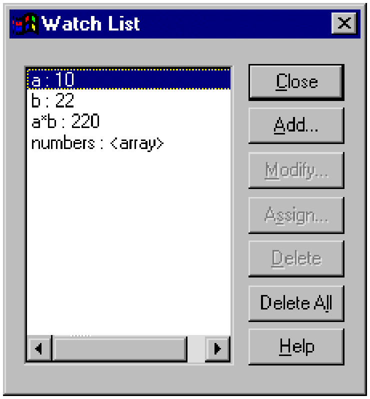
Note: Do not add expressions that assign or increment the value of variables to the Watch List; this can affect test execution.

Viewing Variables in the Watch List

Once variables, expressions, and arrays are added 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:

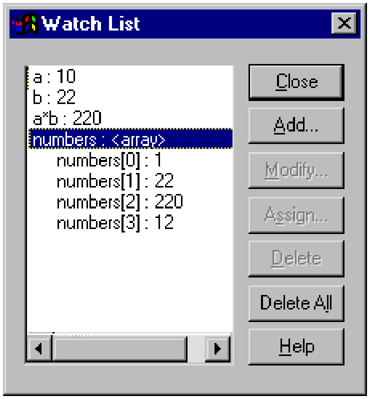
1 Choose Watch List on the Debug menu to open the Watch List dialog box.



```
a : 10
b : 22
a*b : 220
numbers : <array>
```

2 The variables and expressions and arrays are displayed; current values appear after the colon.

3 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.


Watch List

```
a : 10
b : 22
a*b : 220
numbers : <array>
    numbers[0] : 1
    numbers[1] : 22
    numbers[2] : 220
    numbers[3] : 12
```

4 Click Close.

Modifying Variables in the Watch List

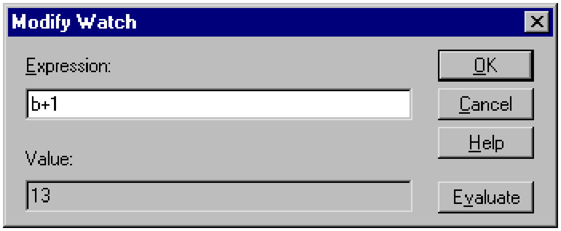
You can modify variables and expressions in the Watch List using the Modify Watch dialog box. For example, you can turn variable *b* into the expression *b + 1*, or you can change the expression *b + 1* into *b * 10*. When you close the Modify Watch dialog box, the Watch List is automatically updated to reflect the new value for the expression.

To modify an expression in the Watch List:

1 Choose Watch List on the Debug menu to open the Watch List dialog box.

2 Select the variable or expression you want to modify.

3 Click Modify to open the Modify Watch dialog box.



Expression: b+1

Value: 13

4 Change the expression in the Expression box as needed. Click Evaluate; the new value of the expression appears in the Value box.

5 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

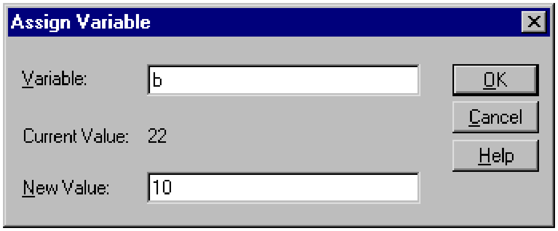
You can assign new values to variables and array elements in the Watch List. Values can be assigned only to variables and array elements, not to expressions.

To assign a value to a variable or an array element:

1 Choose Watch List on the Debug menu to open the Watch List dialog box.

2 Select a variable or an array element.

3 Click Assign to open the Assign Variable dialog box.


Assign Variable

Variable: b

Current Value: 22

New Value: 10

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. Choose Watch List on the Debug menu to open the Watch List dialog box.
2. Select a variable, an expression, or an array to delete.

Note: An array can be deleted 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 to remove the entry from the list.
4. Click Close to close the Watch List dialog box.

To delete all entries in the Watch List:

1. Open the Watch List dialog box.
2. Click Delete All. All entries are deleted.
3. Click Close to close the dialog box.

Configuring WinRunner

Find

?

Customizing WinRunner’s User Interface

You can customize WinRunner’s user interface to adapt it to your testing needs and to the application you are testing.

This chapter describes:

Find

?

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.

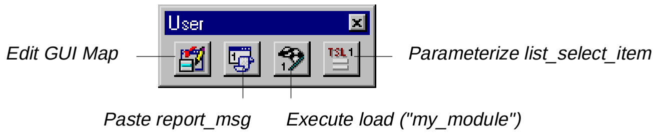
You may find that when you create and run tests, you frequently use the same WinRunner menu commands and insert the same TSL statements into your test scripts. You can create shortcuts to these commands and TSL statements by building a customized toolbar, the User toolbar.

The application you are testing may use softkeys that are preconfigured for WinRunner commands. If so, you can adapt WinRunner’s user interface to this application by using WinRunner’s Softkey utility to reconfigure the conflicting WinRunner softkeys.

Creating the User Toolbar

The User toolbar enables easy access to the commands you most frequently use when testing an application. You can use the User toolbar to:

- Execute 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.



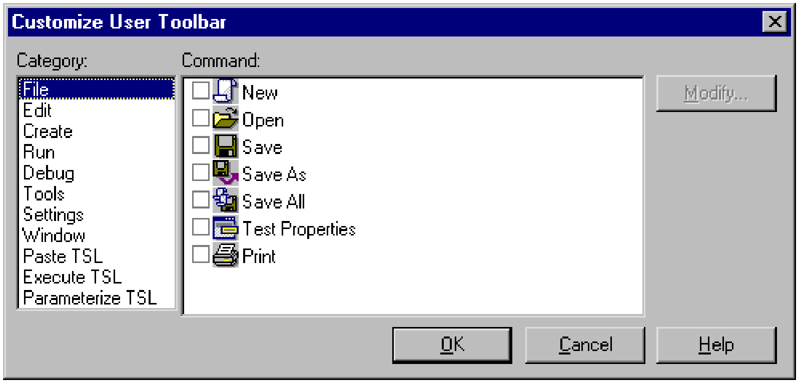
Adding Buttons that Execute Menu Commands

You can add buttons to the User toolbar that execute frequently-used menu commands.

To add a menu command to the User toolbar:

1 Choose Customize User Toolbar on the Settings menu.

The Customize User Toolbar dialog box opens.



2 In the Category box, select a menu.

3 In the Command box, select the check box next to a menu command.

4 Click OK to close the Customize User Toolbar dialog box.

The menu command button is added to the User toolbar.

To remove a menu command from the User toolbar:

1 Choose Customize User Toolbar on the Settings menu to open the Customize User Toolbar dialog box.

2 In the Category box, select the menu.

3 In the Command box, clear the check box next to the menu command.

4 Click OK to close the Customize User Toolbar dialog box.

The menu command button is removed from the User toolbar.

Adding Buttons that Paste TSL Statements

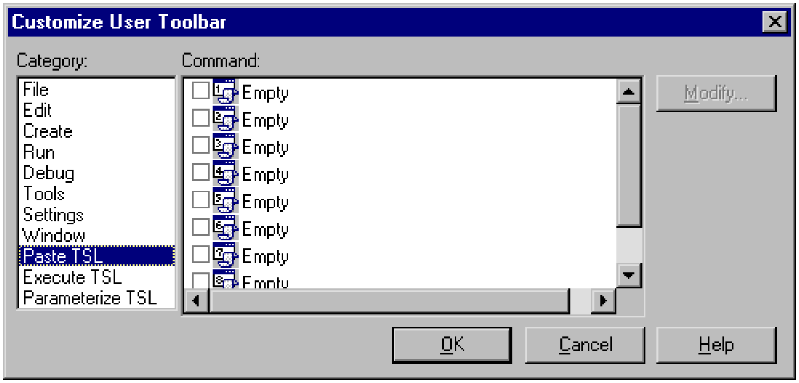
You can add buttons to the User toolbar that paste TSL statements into test scripts. One button can paste a single TSL statement or a group of statements.

To add a button to the User toolbar that pastes TSL statements:

1 Choose Customize User Toolbar on the Settings menu.

The Customize User Toolbar dialog box opens.

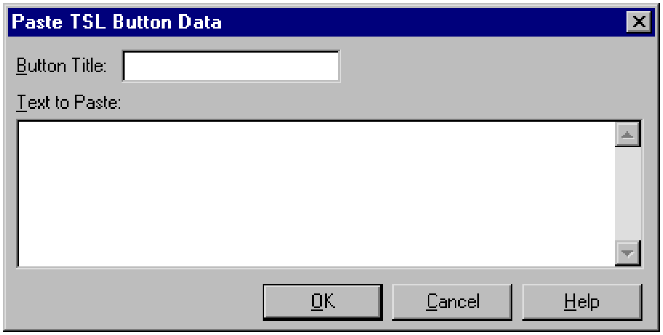
2 In the Category box, select Paste TSL.



3 In the Command box, 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 box, 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 box.

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 Customize User Toolbar on the Settings menu to open the Customize User Toolbar dialog box.

2 In the Category box, select Paste TSL.

3 In the Command box, 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 box.

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 Customize User Toolbar on the Settings menu to open the Customize User Toolbar dialog box.

2 In the Category box, select Paste TSL.

3 In the Command box, 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.

Find
?

Adding Buttons that Execute TSL Statements

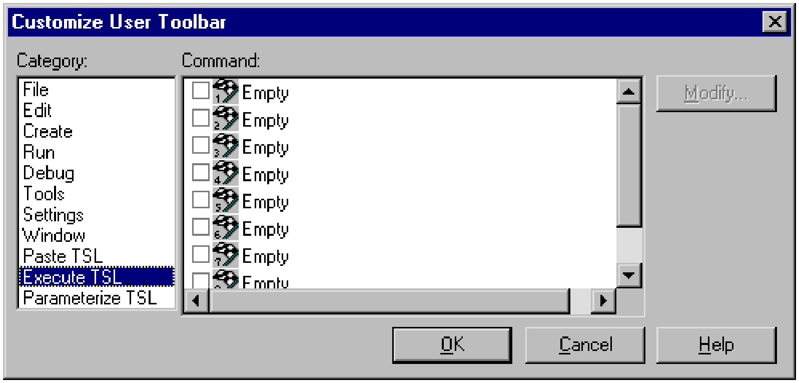
You can add buttons to the User toolbar that execute frequently-used TSL statements.

To add a button to the User toolbar that executes a TSL statement:

1 Choose Customize User Toolbar on the Settings menu.

The Customize User Toolbar dialog box opens.

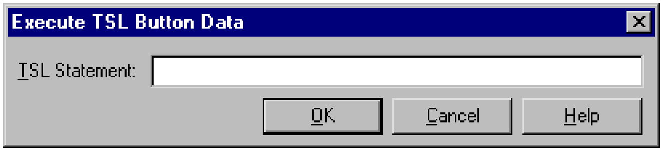
2 In the Category box, select Execute TSL.



3 In the Command box, 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 box.

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 Customize User Toolbar on the Settings menu to open the Customize User Toolbar dialog box.

2 In the Category box, select Execute TSL.

3 In the Command box, 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 Customize User Toolbar on the Settings menu to open the Customize User Toolbar dialog box.

2 In the Category box, select Execute TSL.

3 In the Command box, 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

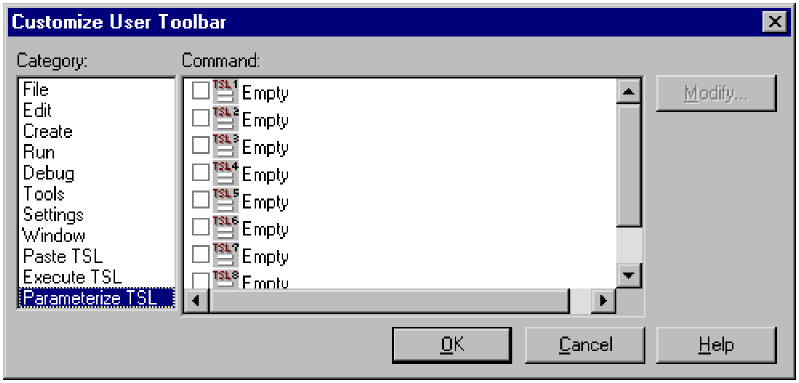
You can add buttons to the User toolbar that enable you to easily parameterize frequently-used TSL statements, and then paste them into your test script or execute them.

To add a button to the User toolbar that enables you to parameterize a TSL statement:

1 Choose Customize User Toolbar on the Settings menu.

The Customize User Toolbar dialog box opens.

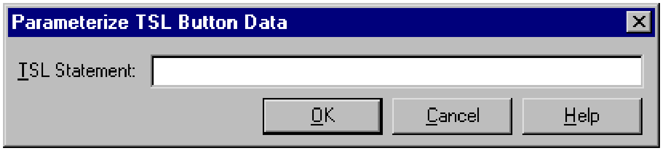
2 In the Category box, select Parameterize TSL.



3 In the Command box, 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 box.

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 Customize User Toolbar on the Settings menu to open the Customize User Toolbar dialog box.
2. In the Category box, select Parameterize TSL.
3. In the Command box, 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 Customize User Toolbar on the Settings menu to open the Customize User Toolbar dialog box.

2 In the Category box, select Parameterize TSL.

3 In the Command box, 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.

Find

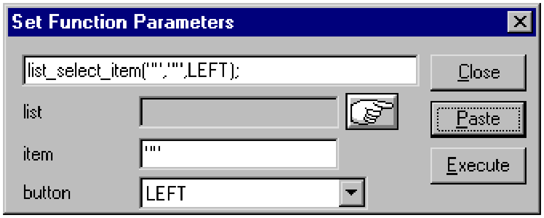
?

Using the User Toolbar

The User toolbar is hidden by default. You can display it by selecting it on the Window 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 commands that appear on the User toolbar by choosing them on the Tools menu.

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);

list:

item: ""

button: LEFT

The Set Function Parameter dialog box varies in its appearance according to the parameters required by a particular TSL function. For example, the **list_select_item** function has three parameters: *list*, *item*, and *button*. For each parameter, you define a value as described below:

- To define a value for the *list* parameter, you click the pointing hand. WinRunner is minimized to an icon, 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.

Accessing TSL Statements on the Menu Bar

All TSL statements that you add to the User toolbar can also be accessed via the Tools menu.

To choose a TSL statement from a menu:

- To paste a TSL statement, click Tools > User Paste TSL > [TSL Statement].
- To execute a TSL statement, click Tools > User Execute TSL > [TSL Statement].
- To parameterize a TSL statement, click Tools > User Parameterize TSL > [TSL Statement].

Find
?

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.

Find

?

Default Settings for WinRunner Softkeys

The following table lists the default softkey configurations and their functions.

Command	**Default Softkey Combination**	**Function**
RUN FROM ARROW	Ctrl Left + F7	Executes the test from the line in the script indicated by the arrow.
RUN FROM TOP	Ctrl Left + F5	Executes the test from the beginning.
STEP	F6	Executes and indicates only the current line of the test script.

Command	Default Softkey Combination	Function
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 test execution 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 execution.
GET TEXT OBJECT	F11	Captures text in an object or window.
GET TEXT AREA	Alt Right + F11	Captures text in a specified area.
MOVE LOCATOR	Alt Left + F6	Records a move_locator_abs statement with the current position (in pixels) of the screen pointer.

Command	Default Softkey Combination	Function
RECORD	F2	Starts test recording. While recording, this softkey toggles between Context Sensitive and Analog modes.
CHECK BITMAP	Ctrl Left + F12	Captures an object or window bitmap.
CHECK BITMAP AREA	Alt Left + F12	Captures an area bitmap.
WAIT BITMAP	Ctrl Left + F11	Instructs WinRunner to wait for a specific object or window bitmap to appear.
WAIT BITMAP AREA	Alt Left + F11	Instructs WinRunner to wait for a specific area bitmap to appear.
CHECK GUI	Ctrl Right + F12	Checks one GUI object or window.
GUI CHECKLIST	F12	Opens the Check GUI dialog box.
INSERT FUNCTION	F8	Inserts a TSL function for a GUI object.
INSERT FUNCTION FROM LIST	F7	Opens the Function Generator dialog box.

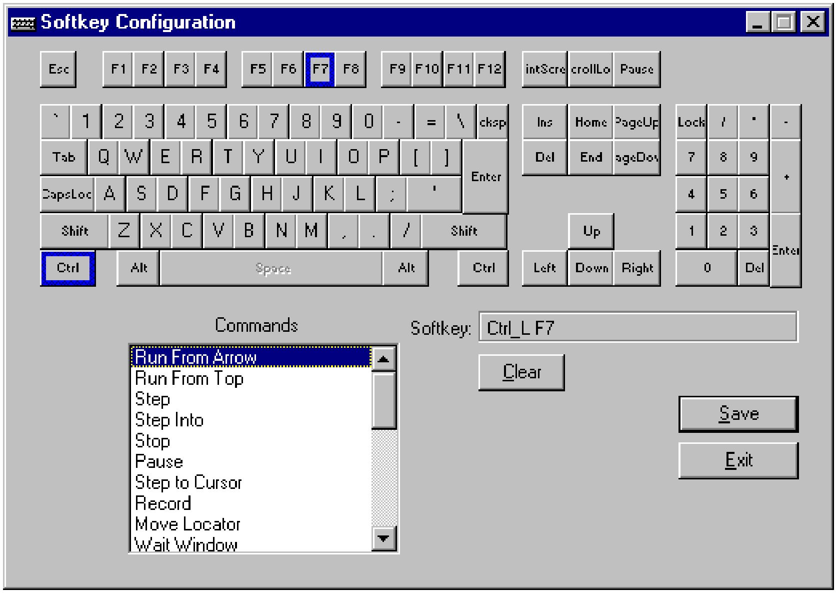
Redefining WinRunner Softkeys

The Softkey Configuration dialog box lists the current softkey assignments and displays an image of a keyboard. To change a softkey setting, you click the new key combination as it appears in the dialog box.

To change a WinRunner softkey setting:

1 Choose Programs > WinRunner > Softkey on the Start menu. The Softkey Configuration dialog box opens.

The Commands list displays all the WinRunner softkeys.


Softkey Configuration
Esc F1 F2 F3 F4 F5 F6 F7 F8 F9 F10 F11 F12 intScre crollLo Pause
Ins Home PageUp Del End ageDov
Lock / * - + Enter
Tab CapsLoc Shift Ctrl Alt Space Alt Ctrl Enter Shift
Up Left Down Right 0 Del
Commands
Clear
Save
Exit

Softkey: Ctrl_L F7

Commands:
- Run From Arrow
- Run From Top
- Step
- Step Into
- Stop
- Pause
- Step to Cursor
- Record
- Move Locator
- Wait Window

2 Click the command you want to change. The current softkey definition appears in the Softkey field; 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 field.

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.

Configuring WinRunner
Setting Global Testing Options

You can control how WinRunner records and runs tests by setting global testing options from the Options dialog box.

This chapter describes:

- **Setting Global Testing Options from the Options Dialog Box**
- **Global Testing Options**

Find

?

About Setting Global Testing Options

WinRunner testing options affect how you record test scripts and run tests. For example, you can set the speed at which WinRunner runs a test, or determine how WinRunner records keyboard input.

Some testing options can be set globally, for all tests, using the Options dialog box. You can also set and retrieve options from within a test script by using the **setvar** and **getvar** functions. 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. For more information about setting and retrieving testing options from within a test script, see Chapter 34, **Setting Testing Options from a Test Script**.

Setting Global Testing Options from the Options Dialog Box

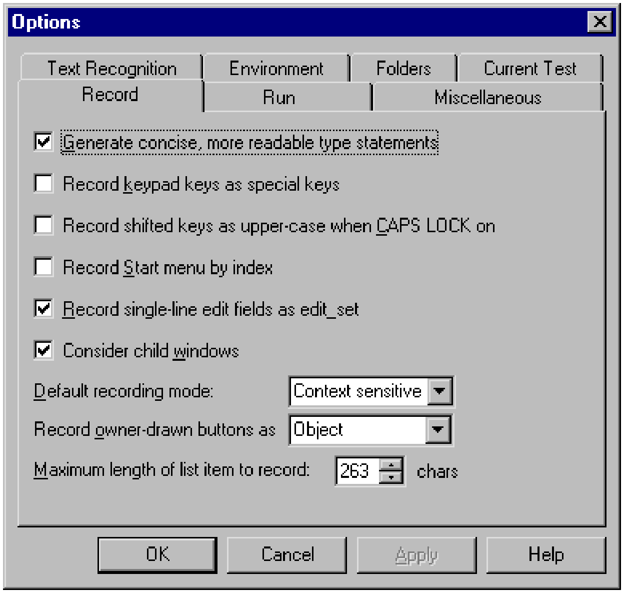
Before you record or run tests, you can use the Options dialog box to modify testing options. The values you set remain in effect for all tests in the current testing session.

When you end a testing session, WinRunner prompts you to save the testing option changes to the WinRunner configuration. This enables you to continue to use the new values in future testing sessions.

To set global testing options:

1 Choose Options from the Settings menu.

The Options dialog box opens. It is divided by subject into seven tabbed pages.



☑ Generate concise, more readable type statements
☐ Record keypad keys as special keys
☐ Record shifted keys as upper-case when CAPS LOCK on
☐ Record Start menu by index
☑ Record single-line edit fields as edit_set
☑ Consider child windows

Default recording mode: Context sensitive
Record owner-drawn buttons as Object
Maximum length of list item to record: 263 chars

2 To choose a page, click a tab.

3 Set an option, as described in **Global Testing Options** on page 574.

4 To apply your change and keep the Options dialog box open, click Apply.

5 When you are done, click OK to apply your change and close the dialog box.

Global Testing Options

The Options dialog box contains the following tabbed pages:

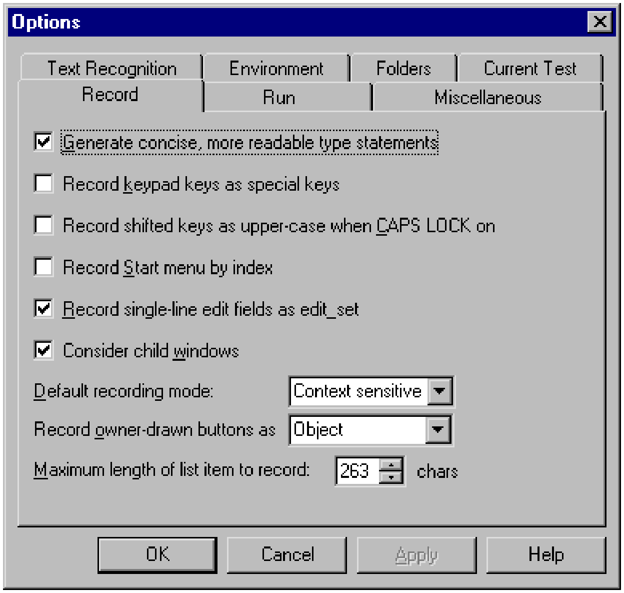
Tab Heading	Subject
Record	Options for recording tests
Run	Options for running tests
Miscellaneous	Options for miscellaneous functions
Text Recognition	Options for recognizing text
Environment	Testing environment options
Folders	Specifying the location of folders for WinRunner files
Current Test	Viewing the settings for the current test

This section lists the global testing options that can be set using the Options dialog box. If an option can also be set within a test script by using the **setvar** function, and retrieved using the **getvar** function, it is indicated below. For more information on the **setvar** and **getvar** functions, see Chapter 34, **Setting Testing Options from a Test Script**.

Find
?

Record Tab

The Record tab options affect how WinRunner records tests.


Options

Text Recognition | Environment | Folders | Current Test | Record | Run | Miscellaneous

- [✓] Generate concise, more readable type statements
- [ ] Record keypad keys as special keys
- [ ] Record shifted keys as upper-case when CAPS LOCK on
- [ ] Record Start menu by index
- [✓] Record single-line edit fields as edit_set
- [✓] Consider child windows

Default recording mode: Context sensitive

Record owner-drawn buttons as Object

Maximum length of list item to record: 263 chars

OK | Cancel | Apply | Help

Generate Concise Type Statements

This option determines how WinRunner generates **type**, **win_type**, and **obj_type** statements in a test script.

☑ When this option is selected, 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 cleared, WinRunner records the pressing and releasing of each key. For example:

```
obj_type (object, "<kShift_L>-a-a+<kShift_L>+");
```

Clear this option if the exact order of keystrokes is important for your test.

(Default = **selected**)

For more information, refer to the **type**, **win_type**, and **obj_type** functions in the *TSL Online Reference*.

Note that you can use the **setvar** and **getvar** functions to set and retrieve the value of the corresponding *key_editing* testing option from within a test script, as described in Chapter 34, **Setting Testing Options from a Test Script**.

Record Keypad Keys as Special Keys

This option determines how WinRunner records pressing keys on the numeric keyboard.

☑ When this option is selected, WinRunner records pressing the NUM LOCK key. It also records pressing number keys and control keys on the numeric keypad as unique keys in the **obj_type** statement it generates. For example:

```
obj_type ("Edit","<kNumLock>")
obj_type ("Edit","<kKP7>")
```

☐ When this option is cleared, WinRunner generates identical statements whether you press a number or an arrow key on the keyboard or on the numeric keypad. WinRunner does not record pressing the NUM LOCK key. It does not record pressing number keys or control keys on the numeric keypad as unique keys in the **obj_type** statements it generates. For example:

```
obj_type ("Edit","7");
```

(Default = **cleared**)

Note: This option does not affect how **edit_set** statements are recorded. When recording using **edit_set**, WinRunner never records keypad keys as special keys.

Record Shifted Keys as Uppercase When CAPS LOCK On

This option determines whether WinRunner records pressing letter keys and the Shift key together as uppercase letters when CAPS LOCK is activated. If WinRunner records pressing letter keys and the Shift key together as uppercase letters when CAPS LOCK is activated, it ignores the state of the CAPS LOCK key when recording and running tests.

☑ When this option is selected, WinRunner records pressing letter keys and the Shift key together as uppercase letters when CAPS LOCK is activated. WinRunner ignores the state of the CAPS LOCK key when recording and running tests.

☐ When this option is cleared, WinRunner records pressing letter keys and the Shift key together as lowercase letters when CAPS LOCK is activated.

(Default = **cleared**)

Record Start Menu by Index

This option determines how WinRunner records on the Windows Start menu in Windows 95 and Windows NT.

☑ When this option is selected, WinRunner records the sequential order in which a menu item appears. For example:

```
button_press ("Start");
menu_select_item ("item_2;item_0;item_4");
```

Select this option when recording the string is expected to fail, e.g. if the name of the menu option is dynamic.

☐ When this option is cleared, WinRunner records the name of the menu item. For example:

```
button_press ("Start");
menu_select_item ("Programs;Accessories;Calculator");
```

(Default = **cleared**)

Note: In Windows 98 and the Microsoft Internet Explorer 4.0 shell, the Start menu does not belong to the menu class, and therefore, this option is not relevant. When WinRunner records on the Start menu in Windows 98 or the Internet Explorer 4.0 shell, it generates a **toolbar_select_item** statement that contains the command strings.

Record Single-Line Edit Fields as edit_set

This option determines how WinRunner records typing a string in a single-line edit field.

☑ When this option is selected, WinRunner records an **edit_set** statement (so that only the net result of all keys pressed and released is recorded). For example, if in the Name field in the Flights Reservation application you type “H”, press Backspace, and then type “Jennifer”, WinRunner generates the following statement:

```
edit_set ("Name","Jennifer");
```

☐ When this option is cleared, WinRunner generates an **obj_type** statement (so that all keys pressed and released are recorded). Using the previous example, WinRunner generates the following statement:

```
obj_type ("Name","H<kBackSpace>Jennifer");
```

(Default = **selected**)

For more information about the **edit_set** and **obj_type** functions, refer to the *TSL Online Reference*.

Consider Child Windows

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.

☑ When this option is selected, WinRunner identifies controls (objects) of a child object whose parent is an object but not a window.

☐ When this option is cleared, WinRunner identifies controls (objects) of a child object whose parent is an object but not a window.

(Default = **cleared**)

Note that you can use the **setvar** and **getvar** functions to set and retrieve the value of the corresponding *enum_descendent_toplevel* testing option from within a test script, as described in Chapter 34, **Setting Testing Options from a Test Script**.

Default Recording Mode

This option determines the default recording mode: either Context Sensitive or Analog. While you are recording your test, you can switch back and forth between recording modes. For more information, see Chapter 3, **Introducing Context Sensitive Testing**.

(Default = **Context sensitive**)

Find
?

Record Owner-Drawn Buttons

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).

(Default = **Object**)

Note that you can use the **setvar** and **getvar** functions to set and retrieve the value of the corresponding *rec_owner_drawn* testing option from within a test script, as described in Chapter 34, **Setting Testing Options from a Test Script**.

Max. Length of List Item Box

This option defines the maximum number of characters that WinRunner can record in a list item name. If the maximum number of characters is exceeded in a List View or TreeView item, WinRunner records that item’s index number. If the maximum number of characters is exceeded in a ListBox or ComboBox, WinRunner truncates the item’s name. The maximum length can be 1 to 263 characters.

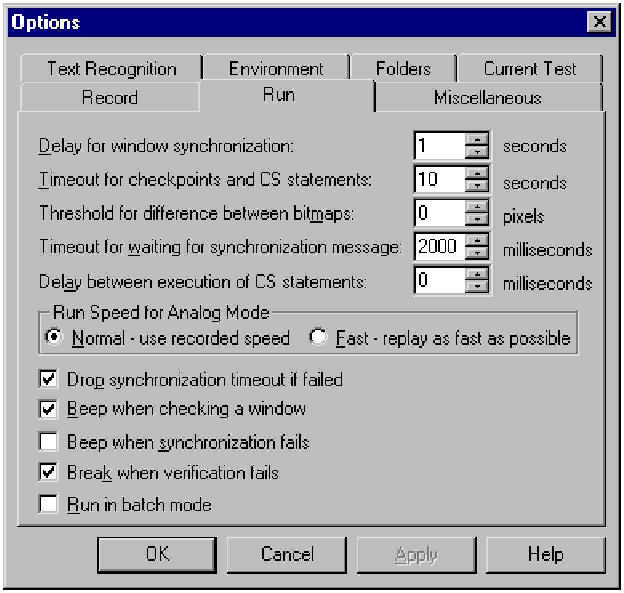
(Default = **263** [characters])

Find

?

Run Tab

The Run tab options affect how WinRunner runs tests.



Delay for window synchronization: 1 seconds
Timeout for checkpoints and CS statements: 10 seconds
Threshold for difference between bitmaps: 0 pixels
Timeout for waiting for synchronization message: 2000 milliseconds
Delay between execution of CS statements: 0 milliseconds

Run Speed for Analog Mode
[✓ Normal - use recorded speed] Fast - replay as fast as possible

- [✓] Drop synchronization timeout if failed
- [✓] Beep when checking a window
- [ ] Beep when synchronization fails
- [✓] Break when verification fails
- [ ] Run in batch mode

Delay for Window Synchronization

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 in the Timeout for Checkpoints and CS Statements box below) is reached.

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.

(Default = **1** [second])

Note that you can use the **setvar** and **getvar** functions to set and retrieve the value of the corresponding *delay* testing option from within a test script, as described in Chapter 34, **Setting Testing Options from a Test Script**.

Find

?

Timeout for Checkpoints and CS Statements

This option sets the global timeout (in seconds) used by WinRunner. 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 in the Delay for Window Synchronization box above).

(Default = **10** [seconds])

Note that you can use the **setvar** and **getvar** functions to set and retrieve the value of the corresponding *timeout* testing option from within a test script, as described in Chapter 34, **Setting Testing Options from a Test Script**.

Threshold for Difference between Bitmaps

This option defines the number of pixels that constitutes the threshold for a bitmap mismatch. When this value is set to 0, a single pixel mismatch constitutes a bitmap mismatch.

(Default = **0** [pixels])

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 34, **Setting Testing Options from a Test Script**.

Timeout for Waiting for Synchronization Message

This option sets the timeout (in milliseconds) that WinRunner waits for synchronization following any keyboard or mouse input when running a test. This option is relevant for all statements run at Fast run speed and for Context Sensitive statements only run at Normal run speed.

(Default = **2000** [milliseconds])

Note that you can use the **setvar** and **getvar** functions to set and retrieve the value of the corresponding *synchronization_timeout* testing option from within a test script, as described in Chapter 34, **Setting Testing Options from a Test Script**.

Note: If synchronization often fails during your test runs, consider increasing the value of this option.

Delay between Execution of CS Statements

This option sets the time (in milliseconds) that WinRunner waits between executing Context Sensitive statements when running a test.

(Default = **0** [milliseconds])

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 34, **Setting Testing Options from a Test Script**.

Drop Synchronization Timeout if Failed

This option determines whether WinRunner drops the synchronization timeout if synchronization fails.

☑ When this option is selected, WinRunner drops the synchronization timeout if synchronization fails.

☐ When this option is cleared, WinRunner does not drop the synchronization timeout if synchronization fails.

(Default = **cleared**)

Note that you can use the **setvar** and **getvar** functions to set and retrieve the value of the corresponding *drop_sync_timeout* testing option from within a test script, as described in Chapter 34, **Setting Testing Options from a Test Script**.

Beep When Checking Window

This option determines whether WinRunner beeps when checking any window during a test run.

☑ When this option is selected, WinRunner beeps when checking any window during a test run.

☐ When this option is cleared, WinRunner does not beep when checking windows during a test run.

(Default = **selected**)

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 34, **Setting Testing Options from a Test Script**.

Find
?

Beep When Synchronization Fails

This option determines whether WinRunner beeps when synchronization fails.

☑ When this option is selected, WinRunner beeps when synchronization fails.

☐ When this option is cleared, WinRunner does not beep when synchronization fails.

(Default = **cleared**)

Note: Select this option primarily to debug a test script.

Note: If synchronization often fails during your test runs, consider increasing the value of the Timeout for Waiting for Synchronization Message option or the corresponding *synchronization_timeout* testing option with the **setvar** function from within a test script.

Note that you can use the **setvar** and **getvar** functions to set and retrieve the value of the corresponding *sync_fail_beep* testing option from within a test script, as described in Chapter 34, **Setting Testing Options from a Test Script**.

Break When Verification Fails

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 selected, WinRunner pauses the test and opens the Run wizard to enable the user to locate the window. If this option is cleared, WinRunner reports an error in the Test Results window and proceeds to run the next statement in the test script.

☑ When this option is selected, 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 run in Verify mode.

☐ When this option is cleared, WinRunner does not pause the test run or display a message whenever verification fails or whenever any message is generated as a result of a context sensitive statement during a test run in Verify mode.

(Default = **selected**)

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 34, **Setting Testing Options from a Test Script**.

Run in Batch Mode

This option determines whether 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 directory, and displays them in one Test Results window.

For example, if a **set_window** statement is missing from a test script, WinRunner cannot find the specified window. If this option is selected 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 cleared 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.

☑ When this option is selected, WinRunner suppresses messages during a test run so that a test can run unattended.

☐ When this option is cleared, WinRunner does not suppress messages during a test run.

(Default = **cleared**)

For more information on suppressing messages during a test run, see Chapter 27, **Running Batch Tests**.

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 34, **Setting Testing Options from a Test Script**.

Run Speed for Analog Mode

This option determines the default run speed for tests run in Analog mode.

Click Normal to run the test at the speed at which it was recorded.

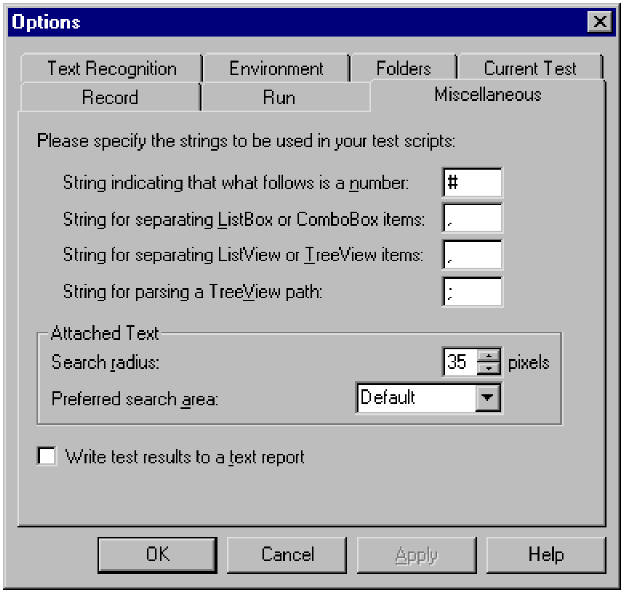
Click Fast to run the test as fast as the application can receive input.

(Default = **Fast**)

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 34, **Setting Testing Options from a Test Script**.

Miscellaneous Tab

The Miscellaneous tab options determine which strings are used by parameters in TSL statements to separate items in a list and identify numbers. Additional options enable you to determine how WinRunner searches and identifies text that is attached to an object.



Please specify the strings to be used in your test scripts:

String indicating that what follows is a number: #

String for separating ListBox or ComboBox items: ,

String for separating ListView or TreeView items: ,

String for parsing a TreeView path: ;

Attached Text

Search radius: 35 pixels

Preferred search area: Default

☐ Write test results to a text report

String Indicating that what Follows is a Number

This option defines the string recorded in the test script to indicate that a list item is specified by its index number. In the following example, the “#” string is used to specify a list item by its index number:

```
set_window ("Food Inventory - Explorer", 3);
list_select_item ("SysTreeView32", "Inventory;Drinks;Soft Drinks");
# Item Number 3;

list_get_items_count("SysListView32", count);
for (i=0; i<count; i++){
	list_select_item ("SysListView32", "#" & i);
	list_get_item ("SysListView32",  i, item);
	list_get_item("ListBox", 0 ,item1);
	if(item != item1)
		tl_step("List Selection Check", FAIL, "Incorrect itemappearing in box for
item: " & item);
}
```

(Default = #)

Note that you can use the **setvar** and **getvar** functions to set and retrieve the value of the corresponding *item_number_seq* testing option from within a test script, as described in Chapter 34, **Setting Testing Options from a Test Script**.

String for Separating ListBox or ComboBox Items

This option defines the string recorded in the test script to separate items in a list box or a combo box.

(Default = **,**)

Note that you can use the **setvar** and **getvar** functions to set and retrieve the value of the corresponding *list_item_separator* testing option from within a test script, as described in Chapter 34, **Setting Testing Options from a Test Script**.

String for Separating ListView or TreeView Items

This option defines the string recorded in the test script to separate items in a list view or a tree view.

(Default = **,**)

Note that you can use the **setvar** and **getvar** functions to set and retrieve the value of the corresponding *listview_item_separator* testing option from within a test script, as described in Chapter 34, **Setting Testing Options from a Test Script**.

String for Parsing a TreeView Path

This option defines the string recorded in the test script to separate items in a tree view path.

(Default = **;**)

Note that you can use the **setvar** and **getvar** functions to set and retrieve the value of the corresponding *treeview_path_separator* testing option from within a test script, as described in Chapter 34, **Setting Testing Options from a Test Script**.

Find

?

Attached Text

The Attached Text box determines how WinRunner searches for the text attached to a GUI object. Proximity to the GUI object is defined by two parameters: the radius that is searched, and the point on the GUI object from which the search is conducted. The closest static text object within the specified search radius from the specified point on the GUI object is that object’s attached text.

Sometimes the static text object that appears to be closest to a GUI object is not really the closest static text object. You may need to use trial and error to make sure that the attached text attribute is the static text object of your choice.

Note: When you run a test, you must use the same setting for the attached text attribute parameters that you used when you recorded the test. Otherwise, WinRunner may not identify your GUI object.

Find
?

Attached Text - Search Radius

This option specifies the radius from the specified point on a GUI object that WinRunner searches for the static text object that is its attached text. The radius can be 3 to 300 pixels.

(Default= **34** [pixels])

Note that you can use the **setvar** and **getvar** functions to set and retrieve the value of the corresponding *attached_text_search_radius* testing option from within a test script, as described in Chapter 34, **Setting Testing Options from a Test Script**.

Find

?

Attached Text - Preferred Search Area

This option specifies the point on a GUI object from which WinRunner searches for its attached text.

Option	Point 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

(Default = **Default**)

Note that you can use the **setvar** and **getvar** functions to set and retrieve the value of the corresponding *attached_text_area* testing option from within a test script, as described in Chapter 34, **Setting Testing Options from a Test Script**.

Note: 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.

Write Test Results to a Text Report

Instructs WinRunner to automatically write test results to a text report, called report.txt, which is saved in the results directory.

☑ When this option is selected, WinRunner automatically writes test results to a text report.

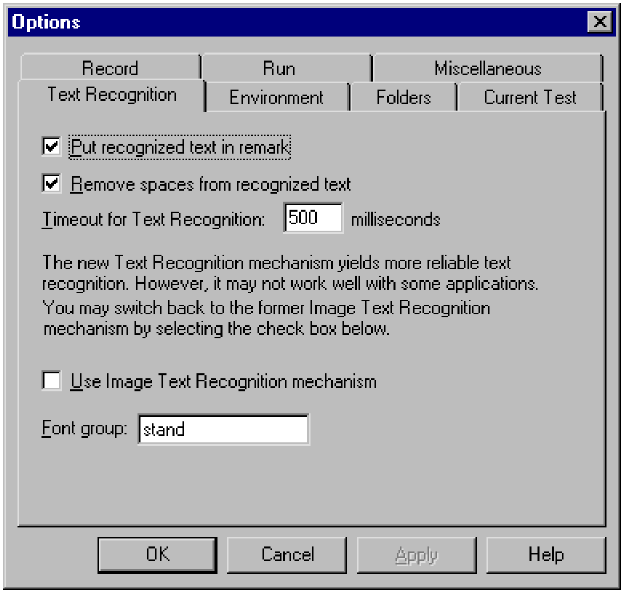
☐ When this option is cleared, WinRunner does not automatically write test results to a text report.

(Default = **cleared**)

Note: A text report of the test results can also be created from the Test Results window by choosing the Tools > Text Report command.

Text Recognition Tab

The Text Recognition tab options affect how WinRunner recognizes text in your application.



[✓] Put recognized text in remark

[✓] Remove spaces from recognized text

Timeout for Text Recognition: 500 milliseconds

The new Text Recognition mechanism yields more reliable text recognition. However, it may not work well with some applications. You may switch back to the former Image Text Recognition mechanism by selecting the check box below.

[ ] Use Image Text Recognition mechanism

Font group: stand

Put Recognized Text in Remark

When you create a text checkpoint, this option determines how WinRunner displays the captured text in the text script.

☑ If this option is selected, WinRunner inserts text captured by a text checkpoint during test creation into the test script as a remark. For example, if you choose Get Text > Object/Window on the Create menu, and then click inside the Fly From text box when Portland is selected, the following statement is recorded in your test script:

```
obj_get_text("Fly From:", text);# Portland
```

☐ If this option is cleared, WinRunner does not insert text captured by a text checkpoint during test creation into the test script as a remark. Using the previous example, WinRunner generates the following statement in your test script:

```
obj_get_text("Fly From:", text);
```

(Default = **selected**)

Remove Spaces from Recognized Text

This option removes leading and trailing blanks in recognized text.

☑ If this option is selected, WinRunner removes leading and trailing blank spaces found in recognized text during test creation from the test script.

☐ If this option is cleared, WinRunner transfers leading and trailing blank spaces found in recognized text during test creation to the test script.

(Default = **selected**)

Note: This option is only relevant for text recognized using the Image Test Recognition mechanism.

Timeout for Text Recognition

This option sets the maximum interval (in milliseconds) that WinRunner waits to recognize text when performing a text checkpoint using Text Recognition during a test run, before using Image Text Recognition, which enables WinRunner to recognize only text whose font is defined in a font group.

(Default = **500** [milliseconds])

Note: If you select the Use Image Text Recognition check box (described in the next section), then the value of this option becomes zero, as timeout has no significance in Image Text Recognition.

Use Image Text Recognition

This option determines the type of text recognition mechanism used by WinRunner when it performs a text checkpoint during a test run. WinRunner can use either Text Recognition or Image Text Recognition: Text Recognition generally yields the most reliable text results, but it does not work well with all applications; Image Text Recognition enables WinRunner to recognize only text whose font is defined in a font group. You should choose this option only if you find that Text Recognition does not work well with the application you are testing.

☑ If this option is selected, WinRunner disables the Text Recognition mechanism and only uses the Image Text Recognition mechanism.

☐ If this option is cleared, WinRunner uses Text Recognition until it is timed out in the interval specified in the Timeout for Text Recognition box (described in the previous section). If Text Recognition fails, WinRunner uses Image Text Recognition.

(Default = **cleared**)

Font Group Box

To be able to use Image Text Recognition (described in the section above), you must choose an active font group. This option sets the active font group for Image Text Recognition. For more information on font groups, see **Teaching Fonts to WinRunner** on page 263.

(Default = **stand**)

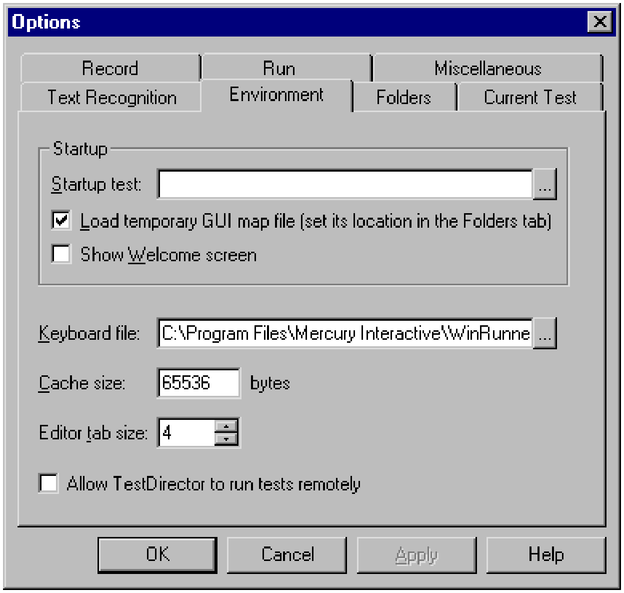
Note that you can use the **setvar** and **getvar** functions to set and retrieve the value of the corresponding *fontgrp* testing option from within a test script, as described in Chapter 34, **Setting Testing Options from a Test Script**.

Find

?

Environment Tab

The Environment tab options affect WinRunner’s testing environment.



Startup

Startup test:

[✓] Load temporary GUI map file (set its location in the Folders tab)

[ ] Show Welcome screen

Keyboard file: C:\Program Files\Mercury Interactive\WinRunne

Cache size: 65536 bytes

Editor tab size: 4

[ ] Allow TestDirector to run tests remotely

Startup Test

This option designates the location of your startup test.

Use a startup test to configure recording, load compiled modules, and load GUI map files when starting WinRunner. Note that you can also set the location of your startup test from the RapidTest Script Wizard.

(Default = **installation directory**)

Note: A startup test can be used in addition to (and not instead of) the initialization (tslinit) test.

Load Temporary GUI Map File

This option determines whether WinRunner automatically loads the temporary GUI map file into the GUI map.

☑ If this option is selected, WinRunner automatically loads the temporary GUI map file when starting WinRunner.

☐ If this option is cleared, WinRunner does not automatically load the temporary GUI map file when starting WinRunner.

(Default = **selected**)

Note: You can set the location of the temporary GUI map file in the Folders tab of the Options dialog box. For more information, see **Temporary GUI Map File** on page 616.

Welcome Screen

This option determines whether the Welcome screen is displayed when starting WinRunner.

☑ If this option is selected, the Welcome screen is displayed when starting WinRunner.

☐ If this option is cleared, the Welcome screen is not displayed when starting WinRunner.

(Default = **selected**)

Keyboard File

This option designates the path of the keyboard definition file. This file specifies the language that appears in the test script when you type on the keyboard during recording.

(Default = **installation directory\dat\win_scan.kbd**)

Cache Size

This option designates the minimum cache size available for garbage collection. If the garbage data is greater than this value, WinRunner frees a block of memory.

(Default = **65536** [bytes])

Editor Tab Size

This option designates the size of the editor tab, which is the number of characters advanced by pressing the Tab key. The maximum size of the editor tab is 128 characters. This option does not affect open tests, but only tests that are open after this parameter is set.

(Default = **4** [characters])

Allow TestDirector to Run Tests Remotely

This option enables TestDirector to run WinRunner tests on your machine from a remote machine. This option also adds the WinRunner Remote Server application to your Windows startup. If the WinRunner Remote Server application is not currently running on your machine, selecting this option starts it. When this application is running, the WinRunner Remote Server icon appears in the status area of your screen.

☑ If this option is selected, TestDirector is allowed to run WinRunner tests from a remote machine. The WinRunner Remote Server application is added to your Windows startup. If the WinRunner Remote Server application is not currently running on your machine, it is started, and its icon appears in the status area of your screen.

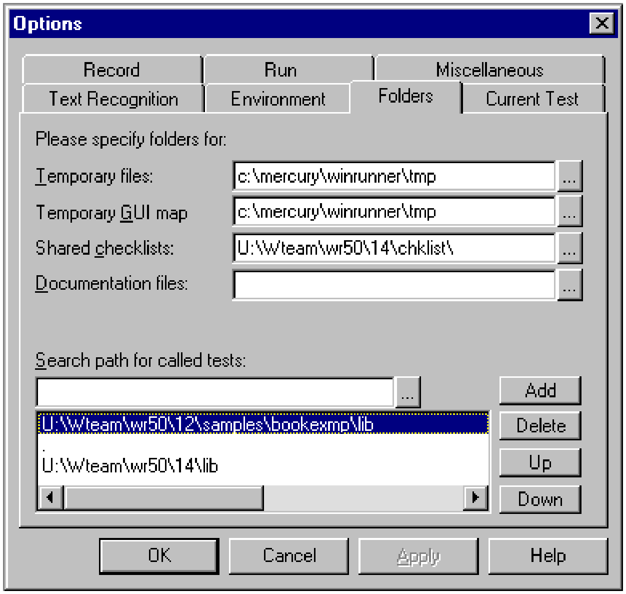
☐ If this option is cleared, TestDirector is not allowed to run WinRunner tests from a remote machine. All WinRunner tests must be run locally. The WinRunner Remote Server application is not part of your Windows startup.

(Default = **cleared**)

For information on running WinRunner tests remotely from TestDirector, refer to your *TestDirector User's Guide*.

Folders Tab

The Folders tab options specify the locations of WinRunner files.



Please specify folders for:

Temporary files: c:\mercury\winrunner\tmp

Temporary GUI map c:\mercury\winrunner\tmp

Shared checklists: U:\Wteam\wr50\14\chklist\

Documentation files:

Search path for called tests:

U:\Wteam\wr50\12\samples\bookexmp\lib
.
U:\Wteam\wr50\14\lib

Temporary Files

This box designates the folder containing temporary tests. To enter a new path, type it in the text box or click Browse to locate the folder. Note that if you designate a new folder, you must restart WinRunner in order for the change to take effect.

(Default = **installation directory\tmp**)

Note that you can use the **setvar** and **getvar** functions to set and retrieve the value of the corresponding *tempdir* testing option from within a test script, as described in Chapter 34, **Setting Testing Options from a Test Script**.

Temporary GUI Map File

This box designates the folder containing the temporary GUI map file (*temp.gui*). If you select the Load Temporary GUI Map File check box in the Environment tab of the Options dialog box, this file loads automatically when you start WinRunner. To enter a new folder, type it in the text box or click Browse to locate it. Note that if you designate a new folder, you must restart WinRunner in order for the change to take effect.

(Default = **installation directory\dat**)

Shared Checklists

This box designates the folder in which WinRunner stores shared checklists for GUI 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**, or **check_gui** statement. To enter a new path, type it in the text box or click Browse to locate the folder. Note that if you designate a new folder, you must restart WinRunner in order for the change to take effect. For more information on shared GUI checklists, see **Saving a GUI Checklist in a Shared Directory** on page 165.

(Default = **installation directory\chklist**)

Note that you can use the **getvar** function to retrieve the value of the corresponding *shared_checklist_dir* testing option from within a test script, as described in Chapter 34, **Setting Testing Options from a Test Script**.

Documentation Files

Designates the folder in which documentation files are stored. To enter a new path, type it in the text box or click Browse to locate the folder.

(Default = **installation directory\doc**)

Search Path for Called Tests

This box determines the paths that 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. The order of the search paths in the list determines the order of locations in which WinRunner searches for a called test.

To add a search path, enter the path in the text box, and click Add. The path appears in the list box, below the text box.

To delete a search path, select the path and click Delete.

To move a search path up one position in the list, select the path and click Up.

To move a selected path down one position in the list, select the path and click Down.

(Default = **installation directory\lib**)

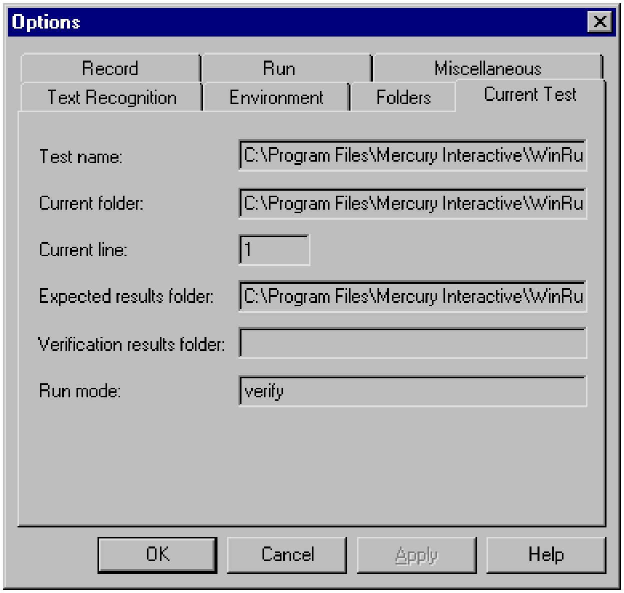
For more information, see Chapter 18, **Calling Tests**.

Note: When WinRunner is connected to TestDirector, you can specify the paths in a TestDirector database that WinRunner searches for called tests. Search paths in a TestDirector database can be preceded by [TD]. Note that you cannot use the Browse button to specify search paths in a TestDirector database: they must be typed directly into the search path box.

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 34, **Setting Testing Options from a Test Script**.

Current Test Tab

The Current Test tab displays the settings for the current test in a read-only view. Note that all the values listed below can also be retrieved by using the **getvar** function.



Test name:	C:\Program Files\Mercury Interactive\WinRu
Current folder:	C:\Program Files\Mercury Interactive\WinRu
Current line:	1
Expected results folder:	C:\Program Files\Mercury Interactive\WinRu
Verification results folder:	
Run mode:	verify

Test Name

This line displays the full path of the current test.

Note that you can use the **getvar** function to retrieve the value of the corresponding *testname* testing option from within a test script, as described in Chapter 34, **Setting Testing Options from a Test Script**.

Current Directory

This line displays the current working directory for the test.

Note that you can use the **getvar** function to retrieve the value of the corresponding *curr_dir* testing option from within a test script, as described in Chapter 34, **Setting Testing Options from a Test Script**.

Current Line Number

This line displays the line number of the current location of the execution arrow in the test script.

Note that you can use the **getvar** function to retrieve the value of the corresponding *line_no* testing option from within a test script, as described in Chapter 34, **Setting Testing Options from a Test Script**.

Expected Results Directory

This line displays the full path of the expected results directory associated with the current test run.

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 34, **Setting Testing Options from a Test Script**.

Results Directory

This line displays the full path of the verification results directory associated with the current test run.

Note that you can use the **getvar** function to retrieve the value of the corresponding *result* testing option from within a test script, as described in Chapter 34, **Setting Testing Options from a Test Script**.

Run Mode

This option displays the current run mode: Verify, Debug, or Update.

Note that you can use the **getvar** function to retrieve the value of the corresponding *runmode* testing option from within a test script, as described in Chapter 34, **Setting Testing Options from a Test Script**.

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

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 Options dialog box. For more information on setting testing options using the Options dialog box, see Chapter 33, **Setting Global Testing Options**.

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	item_number_seq	searchpath
attached_text_search_radius	key_editing	speed
beep	list_item_separator	sync_fail_beep
cs_run_delay	listview_item_separator	synchronization_timeout
delay	min_diff	tempdir
drop_sync_timeout	mismatch_break	timeout
enum_descendent_toplevel	rec_owner_drawn	treeview_path_separator
fontgrp		

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 from the Options dialog box or with another **setvar** statement.

Using the **setvar** function changes a testing option globally, and this change is reflected in the Options dialog box. However, you can also use the **setvar** function to set testing options for all tests, 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* testing option to 20 for a specific test only, insert the following at the beginning of your test script:

```
# Keep the original value of the 'delay' testing option
old_delay = getvar ("delay") ;
setvar ("delay", "20") ;
```

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’ testing option to its original value.
setvar (“delay”, old_delay) ;
```

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
- cs_run_delay
- curr_dir
- delay
- drop_sync_timeout
- enum_descendent_toplevel
- exp
- fontgrp
- item_number_seq
- key_editing
- line_no
- list_item_separator
- listview_item_separator
- min_diff
- mismatch_break
- rec_owner_drawn
- result
- runmode
- searchpath
- shared_checklist_dir
- speed
- sync_fail_beep
- synchronization_timeout
- td_log_dirname
- td_connection
- td_cycle_name
- td_database_name
- td_server_name
- td_user_name
- tempdir
- testname
- timeout
- treeview_path_separator

Find
?

For example:

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

assigns the current value of the run speed to the user-defined variable nowspeed.

Note that some testing options are set by WinRunner and cannot be changed through either **setvar** or the Options dialog box. For example, the value of the testname option is always the name of the current test. Use **getvar** to retrieve this read-only value.

Controlling Test Execution with setvar and getvar

You can use **getvar** and **setvar** together to control test execution without changing global settings. In the following test script fragment, WinRunner checks the bitmap Img1. The **getvar** function retrieves the values of the timeout and delay 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");
d = getvar ("delay");
setvar ("timeout", 30);
setvar ("delay", 3);
win_check_bitmap ("calculator", Img1, 2, 261,269,93,42);
setvar ("timeout", t);
setvar ("delay", d);
```

Note: You can use the **setvar** and **getvar** functions in a startup test script to set environment variables for a specific WinRunner session.

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 an option can also be set or viewed using the Options dialog box, it is indicated below.

attached_text_area

This option specifies the point on a GUI object from which WinRunner searches for its attached text.

Option	Point 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

Option	Point on the GUI Object
Bottom-Left	Bottom-left corner
Left	Midpoint of two left corners

You can use this option with the **setvar** and **getvar** functions.

(Default = **Default**)

Note that you may also set this option using the Attached Text Parameters - Preferred Search Area box in the Miscellaneous tab of the Options dialog box, described in Chapter 33, **Setting Global Testing Options**.

Notes: When you run a test, you must use the same setting for the attached text attribute parameters 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 point on a GUI object that WinRunner searches for the static text object that is its attached text. The radius can be 3 to 300 pixels.

(Default= **34** [pixels])

You can use this option with the **setvar** and **getvar** functions.

Note that you may also set this option using the Attached Text Parameters - Search Radius box in the Miscellaneous tab of the Options dialog box, described in Chapter 33, **Setting Global Testing Options**.

Note: When you run a test, you must use the same setting for the attached text attribute parameters that you used when you recorded the test. Otherwise, WinRunner may not identify the GUI object.

---

batch

This option determines whether 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 directory, and displays them in one Test Results window. For more information on the batch testing option, see Chapter 27, **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.

(Default = **off**)

Note that you may also view this parameter using the Run in Batch Mode check box in the Run tab of the Options dialog box, described in Chapter 33, **Setting Global Testing Options**.

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.

(Default = **on**)

Note that you may also set this option using the corresponding Beep When Checking Window check box in the Run tab of the Options dialog box, described in Chapter 33, **Setting Global Testing Options**.

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.

(Default = **0** [milliseconds])

Note that you may also set this option using the corresponding Delay between Execution of CS Statements box in the Run tab of the Options dialog box, described in Chapter 33, **Setting Global Testing Options**.

curr_dir

This option displays the current working directory for the test.

You can use this option with the **getvar** function.

This option has no default value.

Note that you may also view this parameter using the corresponding Current Directory line in the Current Test tab of the Options dialog box, described in Chapter 33, **Setting Global Testing Options**.

delay

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* testing option) is reached.

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.

(Default = **1** [second])

Note that you may also set this option using the corresponding Delay for Window Synchronization box in the Run tab of the Options dialog box, described in Chapter 33, **Setting Global Testing Options**.

drop_sync_timeout

This option determines whether WinRunner drops the synchronization timeout if synchronization fails.

(Default = **off**)

You can use this option with the **getvar** and **setvar** functions.

Note that you may also view this parameter using the corresponding Drop Synchronization Timeout if Failed check box in the Run tab of the Options dialog box, described in Chapter 33, **Setting Global Testing Options**.

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.

(Default = **off**)

You can use this option with the **getvar** and **setvar** functions.

Note that you may also view this parameter using the corresponding Consider Child Windows check box in the Record tab of the Options dialog box, described in Chapter 33, **Setting Global Testing Options**.

exp

This option displays the full path of the expected results directory associated with the current test run.

You can use this option with the **getvar** function.

This option has no default value.

Note that you may also view this parameter using the corresponding Expected Results line in the Current Test tab of the Options dialog box, described in Chapter 33, **Setting Global Testing Options**.

fontgrp

To be able to use Image Text Recognition (instead of the default Text Recognition), (described in **Use Image Text Recognition** on page 607), you must choose an active font group. This option sets the active font group for Image Text Recognition. For more information on font groups, see **Teaching Fonts to WinRunner** on page 263.

You can use this option with the **setvar** and **getvar** functions.

(Default = **stand**)

Note that you may also set this option using the corresponding Font Group box in the Text Recognition tab of the Options dialog box, described in Chapter 33, **Setting Global Testing Options**.

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.

(Default = #)

Note that you may also set this option using the corresponding String Indicating that what Follows is a Number box in the Miscellaneous tab of the Options dialog box, described in Chapter 33, **Setting Global Testing Options**.

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 Online Reference*.

You can use this option with the **setvar** and **getvar** functions.

(Default = **on**)

Note that you may also set this option using the corresponding Generate Concise Type Statements check box in the Record tab of the Options dialog box, described in Chapter 33, **Setting Global Testing Options**.

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.

This variable has no default value.

Note that you may also view this parameter using the corresponding Current Line Number line in the Current Test tab of the Options dialog box, described in Chapter 33, **Setting Global Testing Options**.

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.

(Default = **,**)

Note that you may also set this option using the corresponding String for Separating ListBox or ComboBox Items box in the Miscellaneous tab of the Options dialog box, described in Chapter 33, **Setting Global Testing Options**.

Find
?

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.

(Default = **,**)

Note that you may also set this option using the corresponding String for Separating ListView or TreeView Items box in the Miscellaneous tab of the Options dialog box, described in Chapter 33, **Setting Global Testing Options**.

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.

(Default = **0** [pixels])

Note that you may also set this option using the corresponding Threshold for Difference between Bitmaps box in the Run tab of the Options dialog box, described in Chapter 33, **Setting Global Testing Options**.

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.

(Default = **on**)

Note that you may also set this option using the corresponding Break when Verification Fails check box in the Run tab of the Options dialog box, described in Chapter 33, **Setting Global Testing Options**.

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.

(Default = **Object**)

Note that you may also view this parameter using the corresponding Record Owner-Drawn Buttons box in the Record tab of the Options dialog box, described in Chapter 33, **Setting Global Testing Options**.

result

This option displays the full path of the verification results directory associated with the current test run.

You can use this option with the **getvar** function.

This option has no default value.

Note that you may also view this parameter using the corresponding Results Directory line in the Current Test tab of the Options dialog box, described in Chapter 33, **Setting Global Testing Options**.

runmode

This option displays the current run mode: Verify, Debug, or Update.

You can use this option with the **getvar** function.

There is no default value for this variable.

Note that you may also view this parameter using the corresponding Run Mode line in the Current Test tab of the Options dialog box, described in Chapter 33, **Setting Global Testing Options**.

searchpath

This option sets the path that 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.

(Default = **installation directory\lib**)

Note that you may also set this option using the corresponding Search Path for Called Tests box in the Folders tab of the Options dialog box, described in Chapter 33, **Setting Global Testing Options**.

Note: When WinRunner is connected to TestDirector, you can specify the paths in a TestDirector database that WinRunner searches for called tests. Search paths in a TestDirector database can be preceded by [TD].

---

shared_checklist_dir

This box designates the folder in which WinRunner stores shared checklists for GUI 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**, or **check_gui** statement. For more information on shared GUI checklists, see **Saving a GUI Checklist in a Shared Directory** on page 165. 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.

(Default = **installation directory\chklist**)

Note that you may also view this parameter using the corresponding Shared Checklists box in the Folders tab of the Options dialog box, described in Chapter 33, **Setting Global Testing Options**.

speed

This option sets the default run *s*peed for tests run in Analog mode. Two speeds are available: *normal* and *fast*.

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.

(Default = **fast**)

Note that you may also set this option using the corresponding Run Speed for Analog Mode option in the Run tab of the Options dialog box, described in Chapter 33, **Setting Global Testing Options**.

sync_fail_beep

This option determines whether WinRunner beeps when synchronization fails.

You can use this option with the **setvar** and **getvar** functions.

(Default = **off**)

Note that you may also set this option using the corresponding Beep When Synchronization Fails check box in the Run tab of the Options dialog box, described in Chapter 33, **Setting Global Testing Options**.

Note: Use this option primarily to debug a test script.

Note: If synchronization often fails during your test runs, consider increasing the value of the *synchronization_timeout* testing option or the corresponding Timeout for Waiting for Synchronization Message option in the Run tab of the Options dialog box.

---

synchronization_timeout

This option sets the timeout (in milliseconds) that WinRunner waits for synchronization following any keyboard or mouse input when running a test. This option is relevant for all statements run at Fast run speed and for Context Sensitive statements only run at Normal run speed.

You can use this option with the **setvar** and **getvar** functions.

(Default = **2000** [milliseconds])

Note that you may also set this option using the corresponding Timeout for Waiting for Synchronization Message box in the Run tab of the Options dialog box, described in Chapter 33, **Setting Global Testing Options**.

Note: If synchronization often fails during your test runs, consider increasing the value of this option.

---

td_connection

This option indicates whether WinRunner is currently connected to TestDirector. (Formerly *test_director*.)

You can use this option with the **getvar** function.

Note that you can connect to TestDirector by choosing TestDirector Connection on the Tools menu. For more information about connecting to TestDirector, see Chapter 37, **Managing the Testing Process**.

(Default = **off**)

td_cycle_name

This option displays the name of the TestDirector test set (formerly known as “cycle”) for the test. (Formerly *cycle*.)

You can use this option with the **getvar** function.

This option has no default value.

Note that you may set this option using the Run Tests dialog box when you run a test set from WinRunner. For more information, see **Running Tests in a Test Set** on page 709. You may also set this option from within TestDirector. For more information, refer to the *TestDirector User’s Guide*.

Find
?

td_database_name

This option displays the name of the TestDirector database to which WinRunner is currently connected.

You can use this option with the **getvar** function.

This option has no default value.

Note that you may set this option using the Database Connection box in the Connection to TestDirector box, which you can open by choosing TestDirector Connection on the Tools menu. For more information, see Chapter 37, **Managing the Testing Process**.

td_server_name

This option displays the name of the TestDirector server (TDAPI) to which WinRunner is currently connected.

You can use this option with the **getvar** function.

This option has no default value.

Note that you may set this option using the Database Connection box in the Connection to TestDirector box, which you can open by choosing TestDirector Connection on the Tools menu. For more information, see Chapter 37, **Managing the Testing Process**.

td_user_name

This option displays the user name for opening the selected TestDirector database. (Formerly *user*.)

You can use this option with the **getvar** function.

This option has no default value.

Note that you may set this option using the User Name box in the Connection to TestDirector box, which you can open by choosing TestDirector Connection on the Tools menu. For more information, see Chapter 37, **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.

(Default = **installation directory\tmp**)

Note that you may also set this option using the corresponding Temporary Files box in the Folders tab of the Options dialog box, described in Chapter 33, **Setting Global Testing Options**.

testname

This option displays the full path of the current test.

You can use this option with the **getvar** function.

This option has no default value.

Note that you may also set this option using the corresponding Test Name line in the Current Test tab of the Options dialog box, described in Chapter 33, **Setting Global Testing Options**.

timeout

This option sets the global timeout (in seconds) used by WinRunner. 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* testing option).

For example, in the statement:

```
win_check_bitmap ("calculator", Img1, 2, 261,269,93,42);
```

when the timeout variable is 10 seconds, this operation takes a maximum of 12 (2+10) seconds.

You can use this option with the **setvar** and **getvar** functions.

(Default = **10** [seconds])

Note that you may also set this option using the corresponding Timeout for Checkpoints and CS Statements box in the Run tab of the Options dialog box, described in Chapter 33, **Setting Global Testing Options**.

treeview_path_separator

This option defines the string recorded in the test script to separate items in a tree view path.

You can use this option with the **getvar** and **setvar** functions.

(Default = **;**)

Note that you may also view this parameter using the corresponding String for Parsing a TreeView Path box in the Miscellaneous tab of the Options dialog box, described in Chapter 33, **Setting Global Testing Options**.

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:

Find

?

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, you perform the following steps:

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 Create > Insert Function > Object/Window command, then 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 Create > Insert Function > From List or press the INSERT FUNCTION FROM LIST softkey.)
2. In the Category box, select “function table”.
3. In the Function Name box, select **generator_add_category**.
4. Click the Args button. 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 673.

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 Create > Insert Function > From List or press the INSERT FUNCTION FROM LIST softkey.)
2. In the Category box, select “function table”.
3. In the Function Name box, select **generator_add_function**.
4. Click the Args button. 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_window***, ***point_object***, ***select_list***, 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 664.
- 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.

Find

?

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.

Find

?

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_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.
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.
select_list:	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

Find
?

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.

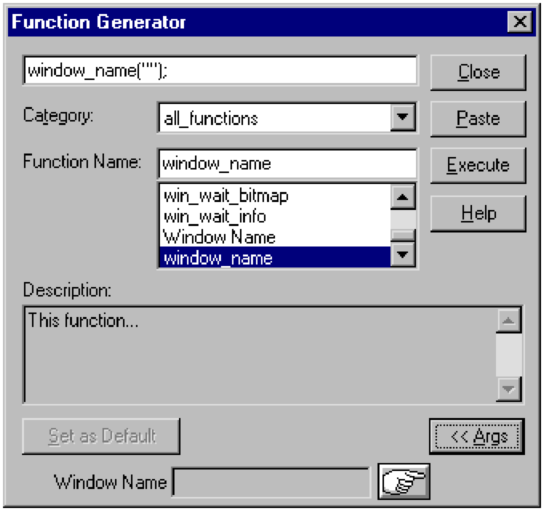
The following are examples of argument definitions that you can include in **generator_add_function statements**. The examples include the syntax of the argument definitions, their representations in the Function Generator, and a brief description of each definition.

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...

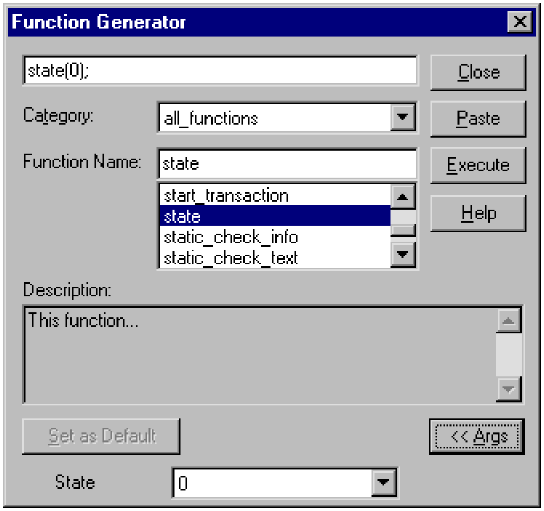
Window Name

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:

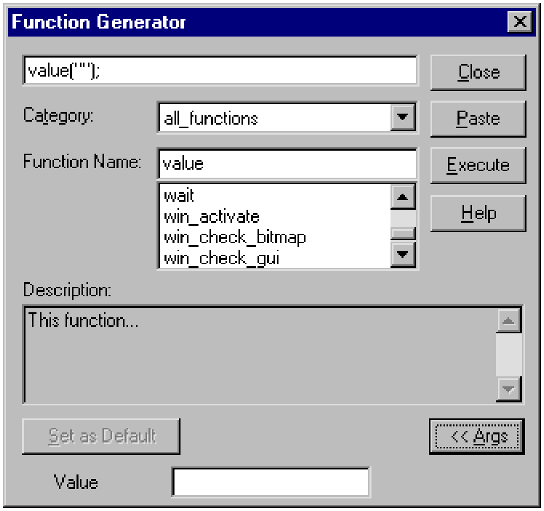


Example 3

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:



value("");

Category: all_functions

Function Name: value

wait
win_activate
win_check_bitmap
win_check_gui

Description:
This function...

Value

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

Any function that you add to the Function Generator should be associated 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 Create > Insert Function > From List or press the INSERT FUNCTION FROM LIST softkey.)
2. In the Category box, select “function table”.
3. In the Function Name box, select **generator_add_function_to_category**.
4. Click the Args button. 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 TSL statement is inserted into your test script. In the following example, the **generator_add_function_to_category** function adds a function called “new_function” to the Function Generator:

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 function **generator_add_subcategory** 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 Create > Insert Function > From List or press the INSERT FUNCTION FROM LIST softkey.)
2. In the Category box, select “function table”.
3. In the Function Name box, select **generator_add_subcategory**.
4. Click the Args button. 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 TSL statement is inserted into your test script. In the following example, the **generator_add_subcategory** function adds a function called “new_function” to the Function Generator:

In the example below, the **generator_add_subcategory** function adds the “my_button” category as a subcategory of the “push_button” category. All “my_button” functions are thereby 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 add a subcategory to a category:

1 Open the Function Generator. (Choose Create > Insert Function > From List or press the INSERT FUNCTION FROM LIST softkey.)

2 In the Category box, select “function table”.

3 In the Function Name box, select **generator_set_default_function**.

4 Click the Args button. 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 TSL statement is inserted into your test script. In the following example, the **generator_set_default_function** function adds a function called “new_function” to the Function Generator:

Find
?

In the example below, the **generator_set_default_function** is used to change the default function of the push button category from **button_check_enabled** to the “check_my_button_label” user-defined 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.

Initializing Special Configurations

By creating *startup tests*, you can automatically initialize special testing configurations each time you start WinRunner.

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 Environment tab in the Options dialog box. For more information on using the Options dialog box, see Chapter 33, **Setting Global Testing Options**.

This chapter describes:

- **Creating Startup Tests**
- **Sample Startup Test**

Creating Startup Tests

It is recommended that you 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 execute 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 operate 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 that you can use the RapidTest Script wizard to create a basic startup test called *mytest* that loads a GUI map file and the application being tested.

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”);
```

Working with TestSuite

Find

?

Working with TestSuite
Managing the Testing Process

Software testing typically involves creating and running thousands of tests. TestSuite's test management tool, TestDirector, can help you organize and control the testing process.

This chapter describes:

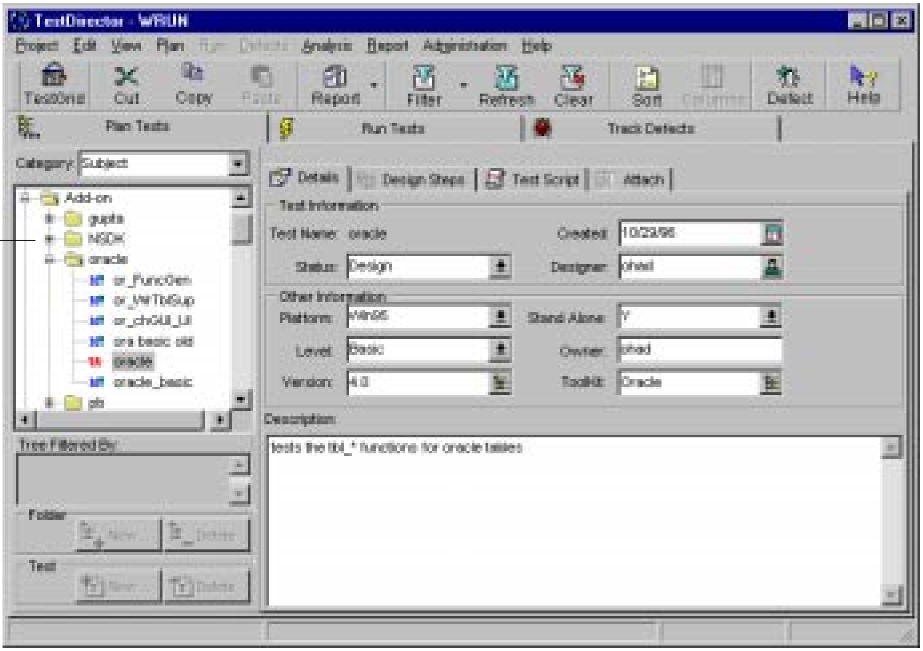
Find

?

Working with TestDirector

TestDirector is a powerful test management tool that enables you to manage and control all phases of software testing. It provides a comprehensive view of the testing process so you can make strategic decisions about the human and material resources needed to test an application and repair defects.

TestDirector divides testing into three modes of operation: planning tests, running tests, and tracking defects. In test planning mode, you begin the testing process by dividing your application into test subjects and building a test plan tree. This is a graphical representation of your test plan, displaying your tests according to the hierarchical relationship of their functions.



Category: Subject

Test Name: oracle
Created: 10/29/98
Status: Design
Designer: chad
Platform: Win95
Stand Alone: Y
Level: Basic
Owner: chad
Version: 4.0
Toolkit: Oracle
Description: tests the tbl_* functions for oracle tables

After you build the test plan tree, you plan tests for each subject. You define steps that describe the operations to perform on the application under test and the expected results of each step. After you define the steps, you decide whether to run the test manually or to automate it. If you decide to automate the test using WinRunner, TestDirector can create a test template for you and launch WinRunner. You then use WinRunner to record and program TSL statements into the template to complete the implementation.

In test run mode, you define test sets. A test set is a group of tests designed to meet a specific testing goal. For example, to verify that the application being tested is functional and stable, you create a sanity test set that checks the application's basic features. You could then create other test sets to test the advanced features.

To build a test set, you select tests from the project database, assign tests to the people who will execute them, and schedule test runs. If your test set contains automated tests, TestDirector can automatically open WinRunner and run the tests. You can run tests on your own computer (locally), or on multiple remote hosts. A host is any computer connected to your network. After WinRunner runs a test in TestDirector, it displays the results and marks the test is marked as passed, failed, or not completed.

In defect tracking mode, you report defects that were detected in the application under test. Information about defects is stored in a defect database. The defects are assigned to developers to be fixed, and then they are tracked until they are corrected.

In all stages of test management, you can create detailed reports and graphs to help you analyze testing data and review the progress of testing on your application.

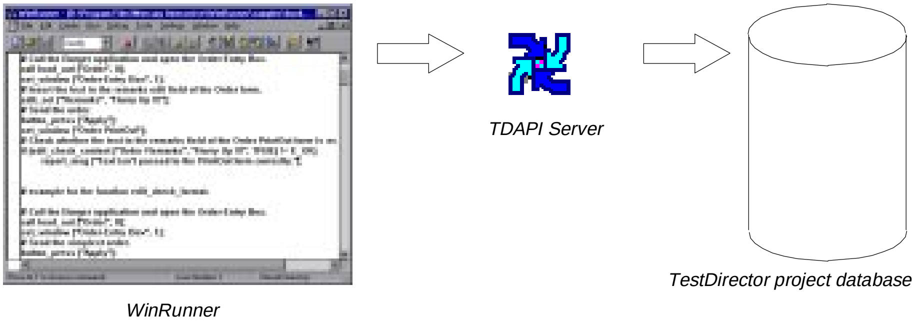
For more information on working with TestDirector, refer to the *TestDirector User's Guide*.

Using WinRunner with TestDirector

TestDirector and WinRunner work together to integrate all aspects of the testing process. In WinRunner, you can create tests and save them in your TestDirector project database. After a test has been run, the results are viewed and analyzed in TestDirector.

TestDirector stores test and defect information in a project database. TestDirector project databases can be either file-based (Microsoft Access) or client/server (Oracle, Sybase, and Microsoft SQL). A file-based database resides on your local file system or in a shared network directory. Client/server databases always reside on a central database server. You create individual project databases within TestDirector. These project databases 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 database, you must connect it to the TDAPI server. This is a program that handles the communication between WinRunner and the TestDirector project database. Note that the TDAPI server usually runs on your TestDirector machine but you can also install it on any computer connected to the network.



When WinRunner is connected to TestDirector, you can save a test by associating it with a subject in the test plan tree, instead of assigning the test to a folder in the file system. This makes it easy to organize tests by subject for your application. When you open a test, you search for it according to its position in the test plan tree. After you run the test, results are sent directly to your TestDirector project database.

Note: The integration of WinRunner 5.0 with TestDirector (described in this chapter) is valid only for TestDirector 5.0.

Note: In order for TestDirector to run WinRunner tests from a remote machine, you must enable the Allow TestDirector to Run Tests Remotely option from WinRunner. By default, this option is disabled. You can enable it from the Environment tab of the Options dialog box (Settings > Options). For more information on setting this option, see Chapter 33, **Setting Global Testing Options**. When this option is enabled, The WinRunner Remote Server application is added to your Windows startup. If the WinRunner Remote Server application is not currently running on your machine, it is started, and its icon appears in the status area of your screen.

Connecting to and Disconnecting from a Database

If you are working with both WinRunner and TestDirector, WinRunner can communicate with your TestDirector project database. You can connect or disconnect WinRunner from a TestDirector project database at any time during the testing process. However, do not disconnect WinRunner from TestDirector while running tests in WinRunner from TestDirector.

The connection process has two stages. First, you connect WinRunner to the TestDirector API server (TdapiSrv). This server handles the connections between WinRunner and the TestDirector project database. Next, you choose the project database you want WinRunner to access. The project database stores tests and test run information for the application you are testing. Note that TestDirector project databases are password protected, so you must provide a user name and a password.

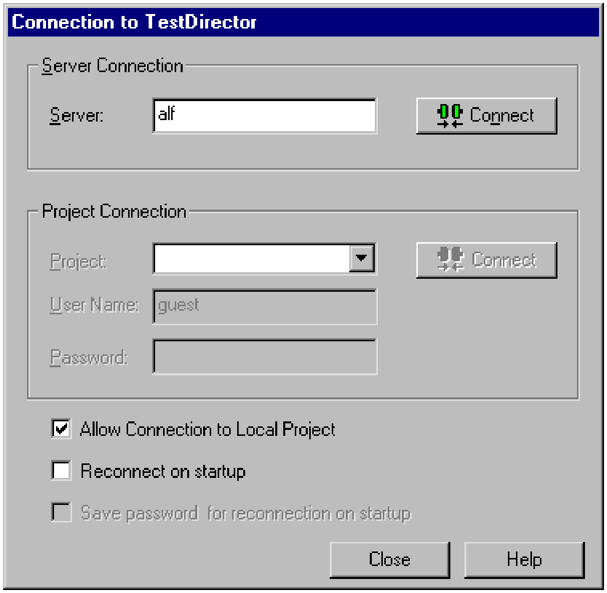
Connecting WinRunner to a TestDirector Server and a Database

You must connect WinRunner to the TestDirector API server before you connect WinRunner to a project database. For more information, see **Using WinRunner with TestDirector** on page 690.

To connect WinRunner to a TestDirector server and a project database:

1 Choose TestDirector Connection on the Tools menu.

The Connection to TestDirector dialog box opens.



2 In the Server Connection section, in the Server box, enter the name of the host where the TestDirector API server (TdapiSrv) runs.

3 Click Connect.

Once the connection to the server is established, the server's name is displayed in read-only format in the Server box.

4 In the Project Connection section, select a TestDirector project database from the Project box.

5 Type a user name in the User Name box.

6 Type a password in the Password box.

7 Click Connect to connect WinRunner to the selected project database.

   Once the connection to the selected project database is established, the project database's name is displayed in read-only format in the Project box.

   To automatically reconnect to the TestDirector server and the selected project database on startup, select the Reconnect on Start Up check box.

   If the Reconnect on Start Up check box is selected, then the Save Password in Registry for Reconnection on Start Up check box is enabled. To save your password for reconnection on startup, select the Save Password in Registry for Reconnection on Start Up check box. If you do not save your password, you will be prompted to enter it when WinRunner connects to TestDirector on startup.

8 Click Close to close the Connection to TestDirector dialog box.

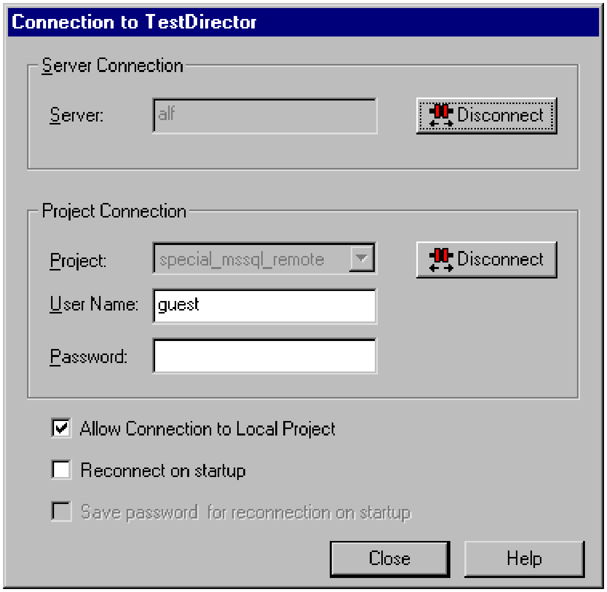
Disconnecting from a TestDirector Database

You can disconnect from a TestDirector project database. This enables you to select a different project database while using the same TestDirector server.

To disconnect WinRunner from a project database:

1 Choose TestDirector Connection on the Tools menu.

The Connection to TestDirector dialog box opens.


Connection to TestDirector

Server Connection

Server: alf

Project Connection

Project: special_mssql_remote

User Name: guest

Password:

[✓] Allow Connection to Local Project

[ ] Reconnect on startup

[ ] Save password for reconnection on startup

2 In the Project Connection section, click Disconnect to disconnect WinRunner from the selected project database.

3 Click Close to close the Connection to TestDirector dialog box.

Disconnecting from a TestDirector Server

You can disconnect from a TestDirector server. This enables you to select a different TestDirector server and a different project database.

To disconnect WinRunner from a project database:

1 Choose TestDirector Connection on the Tools menu.

The Connection to TestDirector dialog box opens.

2 In the Server Connection section, click Disconnect to disconnect WinRunner from the TestDirector server.

3 Click Close to close the Connection to TestDirector dialog box.

Note: If you disconnect WinRunner from a TestDirector server without first disconnecting from a project database, WinRunner's connection to that database is automatically disconnected.

Find

?

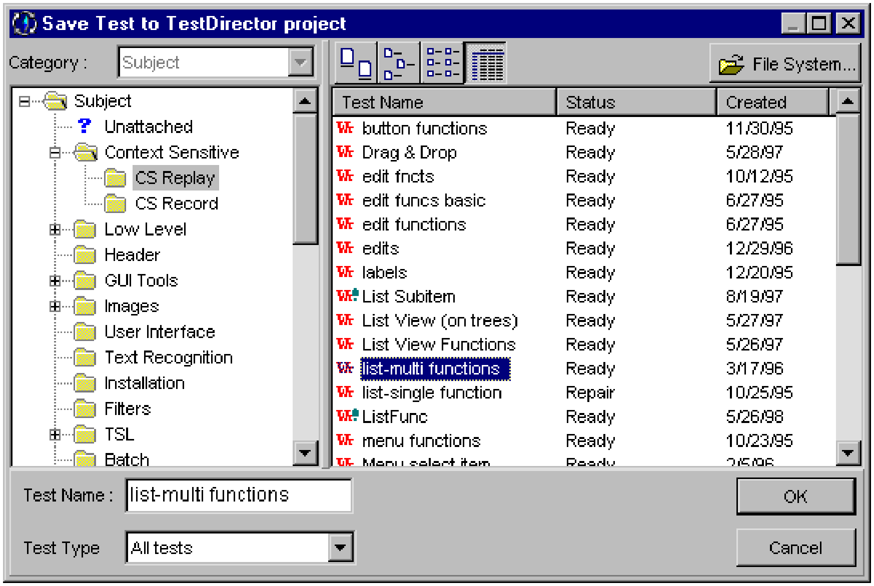
Saving Tests to a Database

When WinRunner is connected to a TestDirector project database, you can create new tests in WinRunner and save them directly to your project database. 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.

To save a test to a TestDirector project database:

1 Choose File > Save As.

The Save Test to TestDirector Project dialog box opens and displays the test plan tree.



Save Test to TestDirector project

Category : Subject

- Subject
  - Unattached
  - Context Sensitive
    - CS Replay
    - CS Record
  - Low Level
  - Header
  - GUI Tools
  - Images
  - User Interface
  - Text Recognition
  - Installation
  - Filters
  - TSL
  - Batch

Test Name	Status	Created
button functions	Ready	11/30/95
Drag & Drop	Ready	5/28/97
edit fncts	Ready	10/12/95
edit funcs basic	Ready	6/27/95
edit functions	Ready	6/27/95
edits	Ready	12/29/96
labels	Ready	12/20/95
List Subitem	Ready	8/19/97
List View (on trees)	Ready	5/27/97
List View Functions	Ready	5/26/97
list-multi functions	Ready	3/17/96
list-single function	Repair	10/25/95
ListFunc	Ready	5/26/98
menu functions	Ready	10/23/95
Menu select item	Ready	2/5/96

Test Name : list-multi functions

Test Type : All tests

Note that the Save Test to TestDirector Project dialog box opens only when WinRunner is connected to a TestDirector project database.

To save a test directly in the file system, click the File System button to open the Save Test dialog box. (From the Save Test dialog box, you may return to the Save Test to TestDirector Project dialog box by clicking the TestDirector button.)

Note: If you save a test directly in the file system, your test will not be saved in the TestDirector project database.

---

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 text 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 to close the dialog box.

Note: To save a batch test, choose WinRunner Batch Test in the Test Type box.

The next time you start TestDirector, the new test will appear in the TestDirector's test plan tree. Refer to the *TestDirector User's Guide* for more information.

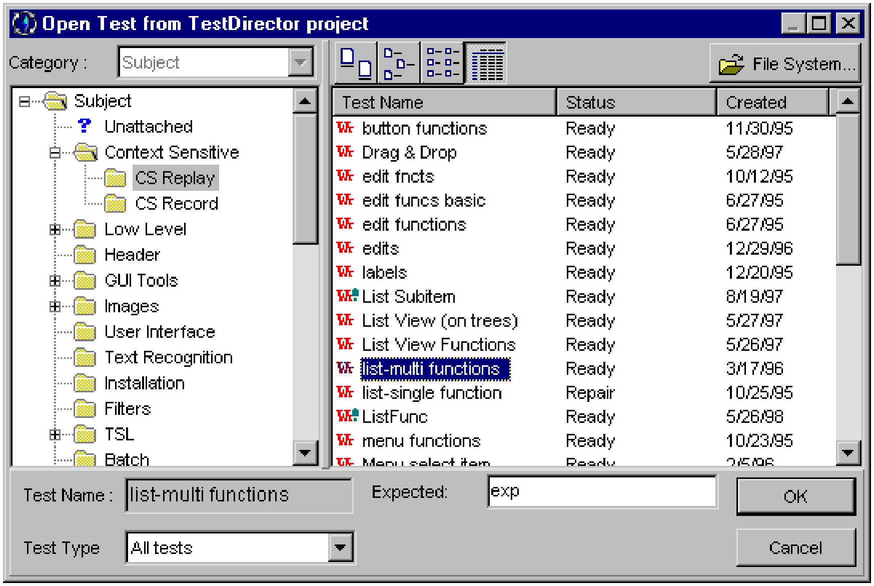
Opening Tests in a Database

If WinRunner is connected to a TestDirector project database, you can open automated tests that are a part of your database. 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 TestDirector project database:

1 Choose File > Open.

The Open Test from TestDirector Project dialog box opens and displays the test plan tree.



- Subject
  - Unattached
  - Context Sensitive
    - CS Replay
    - CS Record
  - Low Level
  - Header
  - GUI Tools
  - Images
  - User Interface
  - Text Recognition
  - Installation
  - Filters
  - TSL
  - Batch

Test Name	Status	Created
button functions	Ready	11/30/95
Drag & Drop	Ready	5/28/97
edit fncts	Ready	10/12/95
edit funcs basic	Ready	6/27/95
edit functions	Ready	6/27/95
edits	Ready	12/29/96
labels	Ready	12/20/95
List Subitem	Ready	8/19/97
List View (on trees)	Ready	5/27/97
List View Functions	Ready	5/26/97
list-multi functions	Ready	3/17/96
list-single function	Repair	10/25/95
ListFunc	Ready	5/26/98
menu functions	Ready	10/23/95

Test Name : list-multi functions
Expected: exp
Test Type All tests

Note that the Open Test from TestDirector Project dialog box opens only when WinRunner is connected to a TestDirector project database.

To open a test directly from the file system, click the File System button to open the Open Test dialog box. (From the Open Test dialog box, you may return to the Open Test from TestDirector Project dialog box by clicking the TestDirector button.)

Note: 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 TestDirector project database.

---

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 in the Test Name list. The test appears in the read-only Test Name box.

4 If desired, enter an expected results directory for the test in the Expected box. (Otherwise, the default directory is used.)

5 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 Test in the Test Type box. For more information on batch tests, see Chapter 27, **Running Batch Tests**.

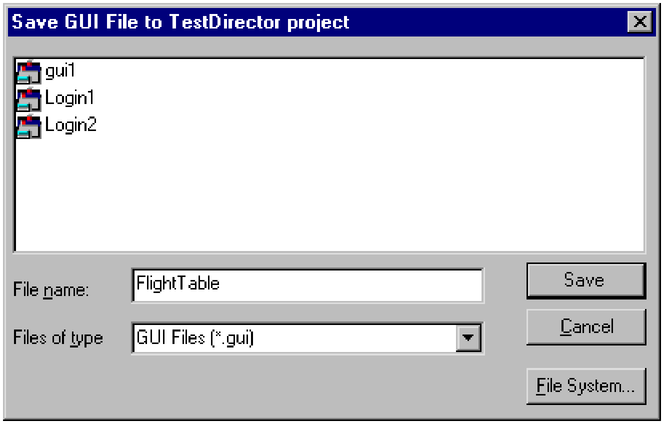
Saving GUI Map Files to a Database

When WinRunner is connected to a TestDirector project database, 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 TestDirector project database are stored together. This facilitates keeping track of the GUI map files associated with tests in your project database.

To save a GUI map file to a TestDirector project database:

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 TestDirector 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 TestDirector Project dialog box opens only when WinRunner is connected to a TestDirector project database.

To save a GUI map file directly to the file system, click the File System button to open the Save GUI File dialog box. (From the Save GUI File dialog box, you may return to the Save GUI File to TestDirector Project dialog box by clicking the TestDirector button.)

Note: If you save a GUI map file directly to the file system, your GUI map file will not be saved in the TestDirector project database.

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.

Opening GUI Map Files in a Database

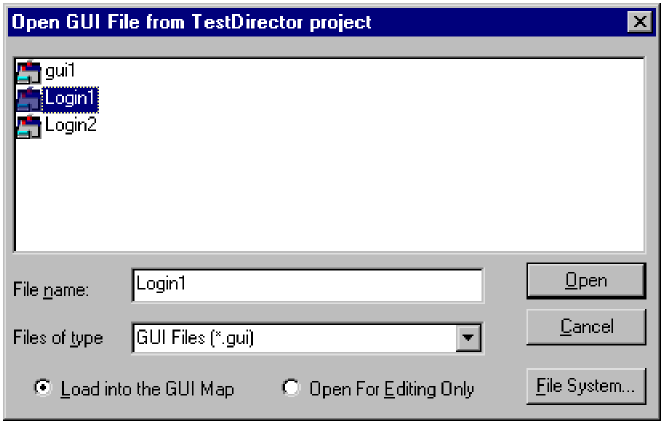
When WinRunner is connected to a TestDirector project database, choose File > Open in the GUI Map Editor to display a list of all GUI map files saved to the open database.

To open a GUI map file saved to a TestDirector project database:

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 TestDirector Project dialog box opens. All the GUI map files that have been saved to the open database are listed in the dialog box.



File name: Login1

Files of type: GUI Files (*.gui)

[✓ Load into the GUI Map] Open For Editing Only

Note that the Open GUI File from TestDirector Project dialog box opens only when WinRunner is connected to a TestDirector project database.

To open a GUI map file directly from the file system, click the File System button to open the Open GUI File dialog box. (From the Open GUI File dialog box, you may return to the Open GUI File from TestDirector Project dialog box by clicking the TestDirector 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 text 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, see Chapter 5, **Editing the GUI Map**.

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

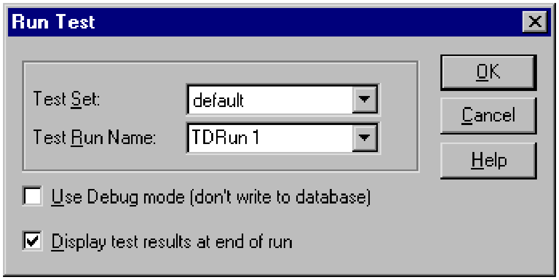
A test set is a group of tests selected to achieve specific testing goals. For example, you can create a test set that tests the user interface of the application or the application's performance under stress. You define test sets when working in TestDirector's test run mode.

If WinRunner is connected to a project database and you want to run tests in the project database from WinRunner, specify the name of the current test set before you begin. When the test run is completed, the test results are stored in TestDirector according to the test set you specified.

To specify a test set and user name:

1 Choose a Run command on the Create menu.

The Run Test dialog box opens.


Run Test

Test Set: default

Test Run Name: TDRun 1

☐ 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 TestDirector.

3 In the Test Run Name box, select a name for this test run, or enter a new name.

   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 TestDirector project database.

   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.

4 Click OK to save the parameters and to run the test.

Running Tests from TestDirector

TestDirector can run WinRunner tests on remote hosts. To enable TestDirector to use your computer as a remote host, you must activate the Allow TestDirector to Run Tests Remotely option.

To enable TestDirector on a remote machine to run WinRunner tests:

1 Choose Settings > Options to open the Options dialog box.

2 Click the Environment tab.

3 Select the Allow TestDirector to Run Tests Remotely check box.

Note: If the Allow TestDirector to Run Tests Remotely check box is cleared, WinRunner tests can only be run locally.

For more information on setting testing options using the Options dialog box, see Chapter 31, “Setting Global Testing Options.”

Viewing Test Results from a Database

If you run tests in a test set, you can view the test results from a TestDirector project database. 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.

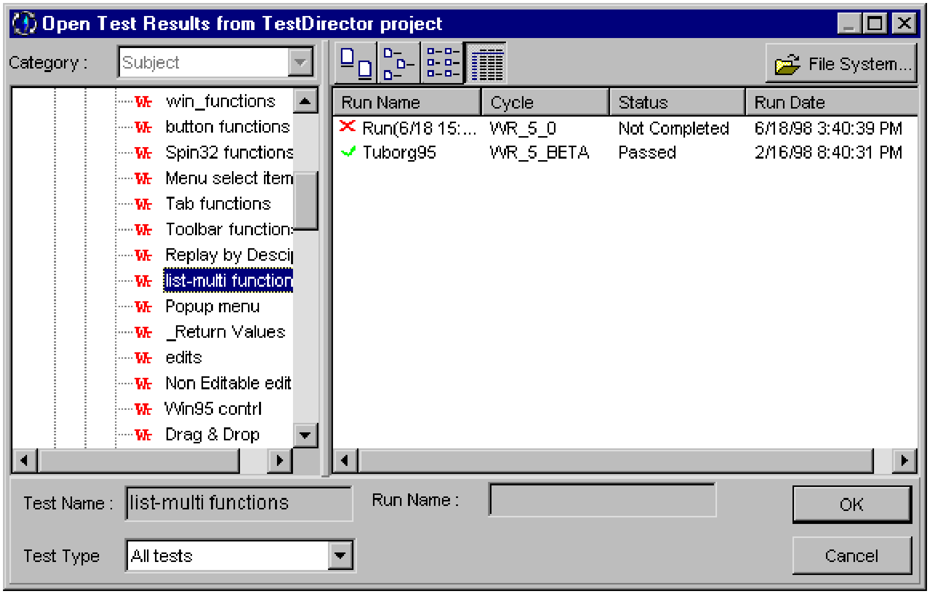
To view test results from a TestDirector project database:

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 TestDirector Project dialog box opens and displays the test plan tree.



Run Name	Cycle	Status	Run Date
Run(6/18 15:...	WR_5_0	Not Completed	6/18/98 3:40:39 PM
Tuborg95	WR_5_BETA	Passed	2/16/98 8:40:31 PM

Test Name : list-multi functions
Run Name :
Test Type: All tests

Note that the Open Test Results from TestDirector Project dialog box opens only when WinRunner is connected to a TestDirector project database.

To open test results directly in the file system, click the File System button to open the Open Test Results dialog box. (From the Open Test Results dialog box, you may return to the Open Test Results from TestDirector Project dialog box by clicking the TestDirector 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. The *Run Name* column displays whether your test run passed or failed and contains the names of the test runs. The *Cycle* 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.

For information about the options in the Test Results window, see Chapter 26, **Analyzing Test Results**.

Using TSL Functions with TestDirector

Several TSL functions facilitate your work with a TestDirector project database by returning the values of fields in a TestDirector project database. In addition, working with TestDirector facilitates working with many TSL functions: when WinRunner is connected to TestDirector, you can specify a path in a TestDirector project database in a TSL statement instead of using the full file system path.

TestDirector Functions

Several TSL functions enable you to retrieve information from a TestDirector project database.

tddb_get_step_value	Returns the value of a field in the "dessteps" table in a TestDirector project database.
tddb_get_test_value	Returns the value of a field in the "test" table in a TestDirector project database.
tddb_get_testset_value	Returns the value of a field in the "testcycl" table in a TestDirector project database.

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 Online Reference*.

Call Statement and Compiled Module Functions

When WinRunner is connected to TestDirector, you can specify the paths of tests and compiled module functions saved in a TestDirector project database 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 TestDirector project database, Subject\Sub1\My_test, you can call it from your test script with the statement:

```
call ("[TD]\\Subject\\Sub1\\My_test");
```

Alternatively, if you specify the “[TD]\Subject\Sub1” search path in the 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 [TD] prefix is optional when specifying a test or a compiled module in a TestDirector project database.

Note: When you run a WinRunner test from a TestDirector project database, you can specify its parameters from within TestDirector, instead of using **call** statements to pass parameters from a test to a called test. For information about specifying parameters for WinRunner tests from TestDirector, refer to the *TestDirector User's Guide*.

For more information on working with the specified Call Statement and Compiled Module functions, refer to the *TSL Online Reference*.

GUI Map Editor Functions

When WinRunner is connected to TestDirector, you can specify the names of GUI map files saved in a TestDirector project database when you use GUI Map Editor functions in a test script.

When WinRunner is connected to a TestDirector project database, WinRunner stores GUI map files in the GUI repository in the database. Note that the [TD] prefix is optional when specifying a GUI map file in a TestDirector project database.

For example, if the My_gui.gui GUI map file is stored in a TestDirector project database, 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 Online Reference*.

Specifying Search Paths for Tests Called from TestDirector

You can configure WinRunner to use search paths based on the path in a TestDirector project database.

In the following example, a **setvar** statement specifies a search path in a TestDirector project database:

```
setvar ( searchpath, [TD]\My_project_database\Subject\Sub1 );
```

For information on how to specify the search path using the Options dialog box, see Chapter 33, **Setting Global Testing Options**. For information on how to specify the search path by using a **setvar** statement, see Chapter 34, **Setting Testing Options from a Test Script**.

Command Line Options for Working with TestDirector

You can use the Windows Run command to set parameters for working with TestDirector. 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 following command line options to set parameters for working with TestDirector:

-td_connection {on | off}

Activates or deactivates WinRunner’s connection to TestDirector.

-td_cycle_name *cycle_name*

Specifies the name of the current test cycle. This option is applicable only when WinRunner is connected to TestDirector.

-td_database_name *database_pathname*

Specifies the active TestDirector project database. WinRunner can open, execute, and save tests in this project database. This option is applicable only when WinRunner is connected to TestDirector.

-td_password

Specifies the password for connecting to a project database in a TestDirector server.

-td_server_name

Specifies the name of the TestDirector server to which WinRunner connects.

-td_user_name *user_name*

Specifies the name of the user who is currently executing a test cycle. (Formerly *user*.)

For more information on using command line options, see Chapter 26, “Running Tests from the Command Line.”

Working with TestSuite
Testing Client/Server Systems

Today’s applications are run by multiple users over complex client/server systems. With LoadRunner, TestSuite’s client/server testing tool, you can emulate the load of real users interacting with your server and measure system performance.

This chapter describes:

Find

?

About Testing Client/Server Systems

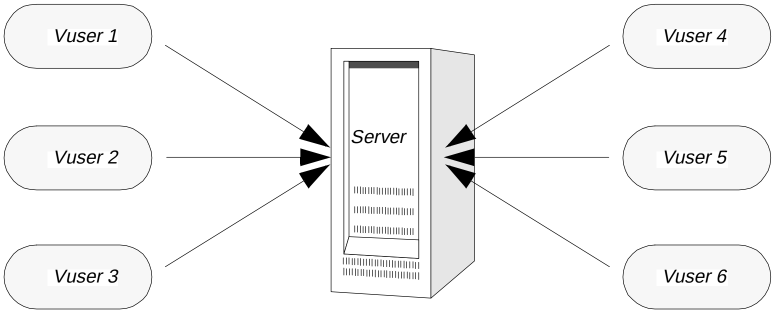
Software testing is no longer confined to testing applications that run on a single, standalone PC. Applications are run in network environments where multiple client PCs or UNIX workstations interact with a central server.

Modern client/server architectures are complex. While they provide an unprecedented degree of power and flexibility, these systems are difficult to test. LoadRunner emulates server load and then accurately measures and analyzes server performance and functionality. This chapter provides an overview of how to use WinRunner together with LoadRunner to test your client/server system. For detailed information about how to test a client/server system, refer to your LoadRunner documentation.

Emulating Multiple Users

With LoadRunner, you emulate the interaction of multiple users (clients) with the server by creating *scenarios*. A scenario defines the events that occur during each client/server 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 the scenario, LoadRunner replaces the human user with a *virtual user or Vuser*. A Vuser emulates the actions of a human user and submits input to the server. A scenario can contain tens, hundreds, or thousands of Vusers.



Load testing your server with LoadRunner

Virtual User (Vuser) Technology

LoadRunner provides a variety of Vuser technologies that enable you to generate server load when using different types of client/server 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; RTE Vusers to operate terminal emulators; TUXEDO Vusers to emulate TUXEDO clients communicating with a TUXEDO application server; Web Vusers to emulate users operating Web browsers.

The various Vuser technologies can be used alone or together, to create effective load testing scenarios.

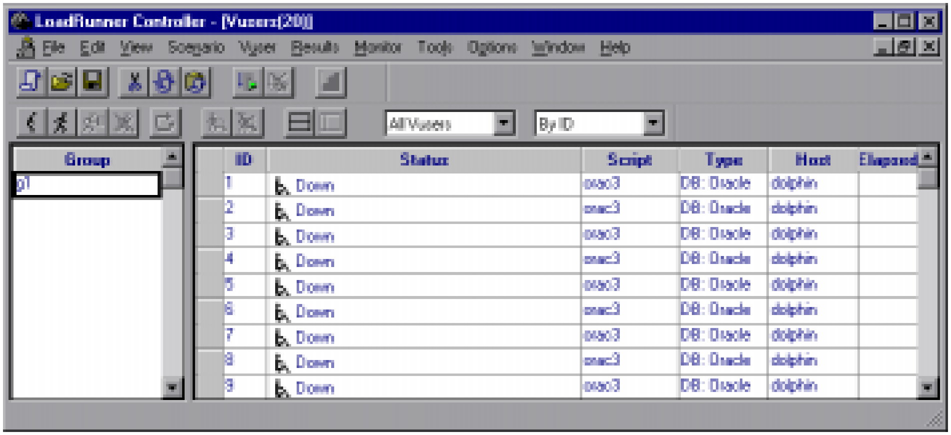
GUI Vusers

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 - [Vusers(20)]
File Edit View Scenario Vuser Results Monitor Tools Options Window Help
All Vusers
By ID
Group
g1

ID	Status	Script	Type	Host	Elapsed
1	Down	orac3	DB: Oracle	dolphin	
2	Down	orac3	DB: Oracle	dolphin	
3	Down	orac3	DB: Oracle	dolphin	
4	Down	orac3	DB: Oracle	dolphin	
5	Down	orac3	DB: Oracle	dolphin	
6	Down	orac3	DB: Oracle	dolphin	
7	Down	orac3	DB: Oracle	dolphin	
8	Down	orac3	DB: Oracle	dolphin	
9	Down	orac3	DB: Oracle	dolphin	

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 client/server testing session, such as the participating Vusers, the scripts they run, and the machines they use to run them (hosts).

3 Run the scenario.

When you run the scenario, LoadRunner distributes the Vusers to their designated hosts. When the hosts 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 run, 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 of 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.

1. Open 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.

Find
?

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).

Find
?

Synchronizing Virtual User 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 of WinRunner's synchronization or object functions. For more information about WinRunner's synchonization functions, see Chapter 14, **Synchronizing Test Execution**.

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. To designate the meeting place, you insert 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 and 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.

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 );
```

Working with TestSuite
Reporting Defects

You can report defects detected in your application using the Remote Defect Reporter.

This chapter describes:

About Reporting Defects

You can report new defects detected in your application using the Remote Defect Reporter. The Remote Defect Reporter opens directly from the WinRunner Test Results window. You provide detailed information about the defect and then send the report to an e-mail address or to a file. Later, you can import the defect information into a TestDirector database, so that the defect can be tracked until it is fixed.

The Remote Defect Reporter window enables you to define general information about the defect and the test in which it was detected. You can also write a detailed description of the defect.

The Remote Defect Reporter looks for setup information in the RDR configuration folder. This information includes available selection lists, field attributes and labels, and the list of available users. The information is stored in three files: Allists.fdb, Fields.fdb, and Users.fdb. Those files are generated automatically by TestDirector.

You can use TestDirector to determine which fields appear in the Remote Defect Reporter by customizing the database. For more information on customizing the database, refer to the *TestDirector Administrator’s Guide*.

Find
?

Setting Up the Remote Defect Reporter

You can choose to send a new defect to an e-mail address or to a file. If you are working on a wide area network, you must report new defects to a mailbox. If you are connected to a local area network, you can achieve best performance by sending new defect reports to a file. In either case, the defect information can later be imported into a TestDirector database.

Note that in order to set up and use the Remote Defect Reporter, you must first do one of the following:

- install TestDirector
- install the Remote Defect Reporter from the TestDirector program group

To set up the Remote Defect Reporter:

1 Make sure that the Test Results window is open.

2 Either choose Report Bug on the Tools menu or click Report Bug on the toolbar.

The first time you open the Remote Defect Reporter window, you are prompted with the following message: “UseMailSystem parameter not defined. Run setup to fix it?”

3 Click Yes.

4 The Remote Defect Reporter setup program opens.

5 Choose the configuration folder, where the files for determining which fields appear in the Remote Defect Reporter are stored. The default location for this folder is the TDPriv folder. To change the location of the configuration folder, click the Browse button, select the desired location, and click OK.

   Click Next to proceed.

6 Choose whether to report defects by e-mail or to a public file: to report defects by e-mail, click Use E-Mail; to store defects in a public file, click Use Public File.

   Click Next to proceed.

7 If you chose to report defects by e-mail, enter the e-mail address for sending defects in the E-Mail Address box. If you chose to report defects to a public file, enter its location in the Report Defects to File box. The name of the file is bugs.fdb.

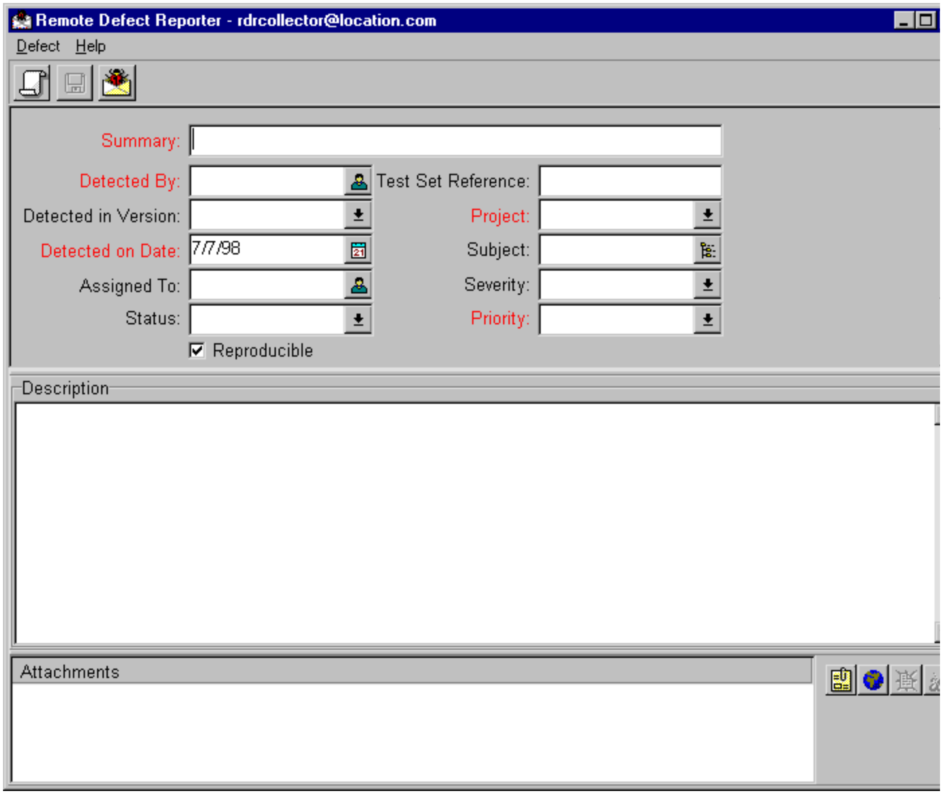
   Click Next to proceed.

8 Click Finish to exit the Remote Defect Reporter setup program.

Note: To change setup options at any time, such as the specified e-mail address or the file location, run the setup program again. To run the setup program, choose Run Setup on the Defect menu in the Remote Defect Reporter window.

The Remote Defect Reporter Window

In the WinRunner Test Results window, click the Report Bug button on the toolbar, or choose Report Bug on the Tools menu. The Remote Defect Reporter window opens.



Summary:

Detected By:

Test Set Reference:

Detected in Version:

Project:

Detected on Date: 7/7/98

Subject:

Assigned To:

Severity:

Status:

Priority:

☑ Reproducible

Description

Attachments

Reporting New Defects

You can report defects detected in your application directly from the WinRunner Test Results window.

To report new defects:

1. In the WinRunner Test Results window, click the Report Bug button on the toolbar, or click Tools > Report Bug. The Remote Defect Reporter window opens.
2. Under Detected By, click the name of the person who detected the defect. Note that the current date appears automatically.
3. Type in a brief summary of the defect.
4. Enter general information about the defect in the General section.
5. In the Description section, type in a detailed description of the defect. You may also type in other information, such as suggestions for working around the defect.
6. Enter test reference information about the defect in the Test Information section.
7. On the Defect menu, choose either the Deliver to File command or the Deliver via E-mail command, depending on the option you chose during setup. Alternatively, click the Deliver to File button or the Deliver via E-mail button on the toolbar. The defect is sent to either an e-mail address or a file. A message appears indicating that the defect was sent.

Note: If you send new defects to an e-mail address and you are not able to send e-mail, an error message may appear. When a defect report is successfully sent via e-mail or to a file, a confirmation message appears on the screen.

Appendix

Find

?

Find

?

The TSL Language

This appendix describes the basic elements of the TSL programming language, including:

Find

?

Variables and Constants

Variables and constants may be either strings or numbers. Declaration is optional; if variables are not declared, their type is determined at run time according to their context.

Variable names can include letters, digits, and underscores (_). The first character must be a letter or an underscore. TSL is case-sensitive; *y* and *Y* are therefore two different characters. Note that names of built-in functions and keywords (such as if, while, switch) cannot be used as variable names.

Find

?

Types of Variables and Constants

TSL supports two types of constants and variables: *numbers* and *strings.* Numbers may be either integer or floating point, and exponential notation is also acceptable. For example, -17, .05, -3e2, and 3E-2 are all legal values.

Strings consist of a sequence of zero or more characters enclosed within double quotes. When a backslash (\) or double-quote (") character appears within a string, it must be preceded by a backslash. Special characters can be incorporated in a string using the appropriate representation:

backspace	\b	vertical tab	\v
carriage return	\r	newline	\n
formfeed	\f	octal number	\ooo
horizontal	\t		

In the case of octal numbers, the zeroes represent the ASCII code of a character. For example, “\126” is equivalent to the letter “v.”

For example, to represent the string “The values are: 12   14   16”, type

```
"\"The values are:\t12\t14\t16\""
```

At a given moment, the value of a constant or variable can be either a string or a number. The TSL interpreter determines the type according to the operation performed. For example:

```
x = 123;
s = x & "Hello";
y = x + 1;
```

Variable *x* is assigned the value *123*. In the second statement, because the operation is concatenation (&), *x* is treated as a string. The interpreted value of *s* is therefore *123Hello*. In the third line, because the operation is addition, *x* is treated as a number. Variable *y* is therefore assigned the value *124.*

In the case of an expression where a mathematical operation is performed on a string, such as

```
"6RED87" + 0
```

the numeric value of the string is the first part of the string that can be evaluated to a number. Here, the numeric value of the expression is 6.

Since relational operators are valid for both strings and numbers, a numeric comparison is always performed if both operands can be evaluated to a number. For instance, in the relational expression below

```
"0.01" == "1e-2"
```

although both constants are written like strings (enclosed within quotation marks), both expressions are also valid numbers so a numeric comparison is performed. But in the next expression

```
"0.01" == "1f-2"
```

the second expression is not a number, so a string comparison is performed.

Undeclared Variables

If a variable is not declared, it is created implicitly when it is assigned or used in an expression. If a variable is not initialized, it is given the string value "" (null) at run time.

All undeclared variables are global, unless they are on the formal Parameter List of a called test. For more information on parameters, see Chapter 18, **Calling Tests**.

Variable Declarations

Note that while constant and variable declarations are optional in tests, they are required in user-defined functions. Variable declarations have the following syntax:

class variable [= *init_expression*];

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

auto: An auto variable can only be declared within a function and is local to that function. It exists only while 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 a Stop command.

public*:* A public variable can only be declared 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.

With the exception of the auto variable, all variables continue to exist until the Stop command is executed. For example, the statement

```
static a=175, b=get_time( ), c = 2.235;
```

defines three variables (a, b, and c), and assigns each an initial value. This value is retained between invocations of the test. The following script segment demonstrates how a static variable can be used so that a message is printed only the first time that the test, T_2, is called.

```
static first = 1;
    pause ("first = " & first);
    if (first == 1) {
    first = 0;
    report_msg ("Test T_2 was called.");
}
```

The following table summarizes the scope, lifetime, and location of the variable declarations for each class:

Declaration	**Scope**	**Lifetime**	**Declare the variable in...**
auto	local	end of function	function
static	local	until stop	function, test, or module
public	global	until stop	test or module
extern	global	until stop	function, test, or module

Constant Declarations

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 the Stop command is executed.

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 be changed only by explicitly making a new constant declaration for TMP_DIR.)

Operators and Expressions

TSL supports six types of operators: arithmetical, concatenation, relational, logical, conditional, and assignment. Operators are used to create expressions by combining basic elements. In TSL, expressions can consist of constants, variables, function calls, and other expressions.

Arithmetical Operators

TSL supports the following arithmetical operators:

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

The result of the modulus operation is assigned the sign of the dividend. For example,

```
7 % -4 = 3
-4.5 % 4 = -0.5
```

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

```
i = 5;
j = i++;
k = ++i;
print(i & j & k);
```

prints the values 7, 5, 7. Note that the increment and decrement operators may be applied only to variables, and not to expressions, such as (a + b).

Concatenation Operator

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

```
x = "ab" & "cd";
```

assigns the string value *abcd* to variable *x*.

Relational Operators

The relational operators used in TSL are:

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

Relational expressions are evaluated to the value 1 if true, and 0 if false. When the value of an expression is null or zero, it is considered false. All other values are considered true.

Strings are compared character by character according to their ASCII value. Letter strings are evaluated in terms of alphabetical order; the string which comes first alphabetically is considered smaller. For instance, “galactic” < “galaxy”.

Logical Operators

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

&&	and
||	or
!	not (unary)

Logical expressions are assigned the value 1 if true, and 0 if false. When the value of an expression is null or zero, it is considered false. All other values are considered true. Logical expressions are evaluated from left to right, and as soon as the value of an expression is determined, interpretation stops. For example, in the expression

```
(g != 0) && (d/g > 17)
```

if the first expression is false, then the second expression is not evaluated.

Conditional Operator

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

expression1 **?** *expression2* **:** *expression3*

expression1 is evaluated first; if it is true, expression2 is evaluated and becomes the value of the expression. If expression1 is false (zero or null), then *expression3* is executed and becomes the value of the expression. In the following statement

```
(g != 0) ? 17 : 18;
```

if the first expression is true (*g* is not equal to zero), then the value of the conditional expression is 17. If the first expression is false, then the value of the conditional expression is 18.

For more information, see **Control-Flow** on page 765.

Assignment Operators

Assignment operators are used to assign values to variables and arrays. All of the binary arithmetical operators have corresponding assignment operators:

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

For example, in the following segment of a test script,

```
for (i=0; i<200; i+=20)
    move_locator_abs(i,i);
```

the value of *i* is incremented by 20 after each repetition of the loop. The mouse pointer is then moved to the new position defined by *i*. For more information about for loops see **Control-Flow** on page 765.

Precedence and Associativity of Operators

The rules of precedence and associativity determine the order in which operations are performed when more than one operator appears in an expression. Operators with higher precedence are interpreted before operators with lower precedence. For example, multiplication is performed before addition.

When more than one operator of the same level of precedence appears in an expression, the associativity indicates the order in which they are interpreted. For example, in

```
x / 2 + i - q
```

division is performed first. Addition is performed before subtraction because the associativity of these operators, which have the same level of precedence, is left to right.

The following table lists the precedence, in descending order, and the associativity of operators:

Operator (in order of precedence)	**Associativity**
( ) (parentheses)	none
++ --	none
^ **	right to left
! - + (unary)	none
* / %	left to right
+ - (binary)	left to right
&	left to right
< <= > >= == !=	none
in (array operator)	none
&&	left to right
||	left to right
?	right to left
= += -= *= /= %= ^= **=	right to left

Statements

Any expression followed by a semicolon is a statement. A statement can continue beyond one line.

In a control-flow structure, a single statement can be replaced by a group of statements, or block. Statements are grouped by enclosing them within curly brackets { }. Each individual statement within brackets is followed by a semicolon, but the brackets themselves are not. This is illustrated below:

```
for (i = 0; i < 10; i++) {
    st = "Iteration number " & i;
    type (st);
}
```

Control-Flow

TSL control-flow statements include:

- *if/else* and *switch* for decision-making
- *while*, *for*, and *do* for looping
- *break* and *continue* for loop modification

If/Else Statement

TSL provides an *if/else* statement for decision-making. The *else* clause is optional. The syntax of this statement is:

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

The *expression* is evaluated; if the value of the *expression* is true (nonzero or non-null), *statement1* is executed; if the value is false (zero or null), and the [else *statement2*] clause is included, *statement2* is executed.

When if statements are nested, the TSL interpreter associates each *else* with the if that appears closest to it. For example, a statement such as:

```
if (b1) if (b2) s1; else s2;
```

is interpreted as follows:

```
if (b1) {
    if (b2)
        s1;
    else
        s2;
}
```

Switch Statement

The *switch* statement provides the mechanism for a multi-way decision. The syntax of this structure is:

```
switch ( expression )
{
    case case_expr1:
        statement(s)
    case case_expr2:
        statement(s)
    case case_exprn:
        statement(s)
[ default: statement(s) ]
}
```

The *switch* statement consecutively evaluates each of the enumerated case expressions (*case_expr1, case_expr2,.... case_exprn*), until one is found that equals the initial *expression.* If no case expression is equal to the specified *expression*, then the optional default statements are executed.

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 within a case to end the loop. For example:

```
switch (a) {
case"xyz":
    b = a & "tw";
    break;
case"uv":
    pause ("hello");
    x = a;
    break;
default:
    x = a;
}
```

Note that while the initial expression can be any regular expression, case expressions can only be constants or variables.

Looping Statements

TSL provides several statements that enable looping.

while (*expression* **)**
    statement

While the *expression* is true, the *statement* is repeatedly executed. At the start of each repetition of the loop, the *expression* is evaluated; if it is true (nonzero or non-null), the *statement* is executed, and the *expression* is re-evaluated. The loop ends when the value of the *expression* is false. For example,

```
i = 1;
while (i < 21)
    type (i++);
```

types the value of *i* 20 times.

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

First, *expression1* is implemented as the starting condition. While *expression2* is true, the *statement* is executed, and *expression3* is evaluated. The loop repeats until *expression2* is found to be false.

This statement is equivalent to:

```
expression1                 # state initial condition
while (expression2) {       # while this is true
    statement               # perform this statement and
    expression3             # evaluate this expression
}
```

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

```
for (i=1; i<21; i++)
    type (i);
```

Note that if *expression2* is missing, it is always considered true, so that

```
for (i=1;i++)
    type (i);
```

is an infinite loop.

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

The *statement* is executed and then the *expression* is evaluated. If the *expression* is true, then the cycle is repeated. This statement differs from the *while* and *for* statements in that the *expression* is evaluated at the end. Therefore, the loop is always executed at least once. For example, in the following statement,

```
i = 20;
do
    type (i++);
while (i < 17);
```

the structure of the loop ensures that the value of *i* is typed at least once.

Loop Modification

The following statements can be used to exit a loop or to jump to the next iteration.

break;

The *break* statement causes an exit from within a loop. If loops are nested, *break* affects the innermost *for*, *while*, or *do* loop that encloses it.

For example, a *for* loop where *expression2* is undefined can be terminated using *break*:

```
for (i = 1; i++) {
    type (i);
    if (i > 29)
        break;
}
```

continue;

The *continue* statement causes the next cycle of the loop to begin. In a *do/while* loop, execution resumes with the test expression. In a *for* loop, execution resumes with *expression3*. For example:

```
for (i = 1; i<=300; i++) {
    if (i % 3 != 0) {
        continue; # to next number
    }
    ...             # long processing
    type(i & "<kReturn>");
}
```

Here, a certain process should only be performed on every third number. Therefore, if *i* cannot be divided equally by three, execution continues with the next iteration of the loop.

Arrays

TSL supports associative arrays. Arrays in TSL are unique in that:

- Array declaration and initialization are optional.
- Each element has a user-defined string subscript.

Rather than arrays of fixed length with numeric subscripts, TSL arrays contain an undefined number of elements, each with a user-defined string subscript. For example, the statement

```
capitals["Ohio"] = "Columbus";
```

assigns the value "Columbus" to the element with subscript "Ohio" in the array *capitals*. If array elements are not declared, they are created the first time they are mentioned and the order of the elements in the array is not defined. Any uninitialized array element has the numeric value zero and the string value null ("").

Arrays can be used to store both numbers and strings. In the following test script, an array is used to store a series of dates and times:

```
for (i=0; i<5; i++) {
    date = time_str();
    date_array[i] = date;
    wait(5);
}
```

Here, each array element includes the date and time of the call to the **time_str** function. The subscript of the array element is the value of *i*.

Array Declaration

Array declaration is optional within a test but required within user-defined functions (initialization is optional). Using the following syntax, you can define the class and/or the initial expression of an array. Array size need not be defined in TSL.

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

The array *class* may be any of the classes listed under Variable Declarations. The *init* expression can take one of two formats: C language syntax, or a string subscript for each element.

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, as in C, arrays with the class *auto* cannot be initialized.

Find

?

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 interpretation or 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
```

Array Initialization

Arrays are initialized once during a test run. The TSL interpreter maintains the original initialization values throughout the test run. If you edit an array's initialization values, the new values will not be reflected during test execution. To reset the array with new initialization values, perform one of the following:

- stop/abort the test run
- define the array elements explicitly

When you stop the test run, all of the script's variables are destroyed. The next time you execute the script, the array is initialized with the new values.

Alternatively, you can explicitly define an array's elements. When you assign a value to each array element, you ensure that the array is updated with the new values for each test run. In the following example, the regular array initialization is replaced with explicit definitions:

Regular Initialization	Explicit Definitions
public array[] = {1,2,3};	array[0] = 1;
	array[1] = 2;
	array[2] = 3;

Multidimensional Arrays

TSL supports multidimensional arrays such as a[i,j,k]. Multidimensional arrays can be used like records or structures in other languages. For example, the following script uses a multidimensional array to store the date and time:

```
for (i = 0;i < 10; i++) {
    date=time_str();
    split(date,array," ");
    multi_array[i, "day"] = array[1];
    multi_array[i, "time"] = array[4];
    wait(5);
}
```

TSL simulates multidimensional arrays using one-dimensional arrays. The element multi_array[i_1, i_2,...i_n] is stored in the one-dimensional array called multi_array, in the element [i_1 & SUBSEP & i_2 & SUBSEP... & i_n]. (The variable SUBSEP has the initial value “\034,” but this value may be changed.)

Multidimensional arrays can also be declared and initialized, as described above. For example, a multidimensional array could be initialized as follows:

```
static rectangles [ ] = {
     {153, 212, 214, 437},
     {72, 112, 88, 126},
     {351, 312, 399, 356}
    }
```

The in Operator

The *in* operator is used to determine if a subscript exists in an array.

subscript **in** array;

returns the value 1 if the subscript exists, and 0 if it does not. It can be used in a conditional statement, like the one below which checks whether the element with the subscript *new* exists in the array *menu_array:*

```
if ("new" in menu_array)
```

The operator *in* should be used rather than the following statement:

```
if (menu_array["new"] != "")...
```

because this statement causes the element to be created, if it does not already exist. (Recall that array elements are created the first time they are mentioned.)

The *in* operator can also be used for multidimensional arrays. The subscript of the element is enclosed in parentheses, as in the following statement:

```
if (("new.doc", 12) in multi_array)...
```

for (element in array) statement

Causes the *element* to be set to the subscript of each element in the *array*. The statement is executed once for each element of the array, and the loop is terminated when all elements have been considered. The order in which the subscripts are read is undefined. The sample script below reads an array for which each element is a date and time string. A *for* loop is used to print to the screen each of the elements of the array.

```
for (i in date_array)
    print ("the date was " & date_array[i]);
```

Specifying a Starting Subscript

TSL allows you to assign values to array elements starting from a specific subscript number. You specify the starting subscript in the array initialization. Remember that the array subscripts are zero-based—the first subscript number is 0.

abc[ ] = {*starting subscript = value1, value2, value3...* }

For example, if the array size is ten, you can assign values to the last five elements of the array:

```
public abc[ ] = {5 = 100,101,102,103,104}
```

As a result, the abc array receives the following values:

```
abc[5]=100
abc[6]=101
abc[7]=102
abc[8]=103
abc[9]=104
```

Array Functions

TSL provides two array functions: **delete** and **split**. The **delete** function removes an element from an array; **split** splits a string into fields and stores the fields in an array. Note that since TSL arrays are associative, deleting one element does not affect any other element. For instance, if you delete the element a[2] from an array with three elements, a[1] and a[3] will not be affected. For details, see the alphabetical reference.

Find

?

Input-Output

TSL provides a number of built-in functions that allow you to read and write to files or to the screen.

For XRunner and other UNIX products, the **print** and **printf** functions are used to write to the screen or to a file. The **print** function prints simple expressions, while the **printf** function generates formatted output. Output can be printed to the screen or written to a file using the appropriate redirection operator. The **close** function closes a file that was opened in response to a **print** or **printf** statement. The **getline** function is used to read a line from a file to a variable. The **sprintf** function returns a formatted string to a variable.

For WinRunner and other PC products, use the **file_open** function to open a file for reading and writing. The **file_printf** function writes to a file, and **file_getline** reads from a file. The **file_close** function closes a file that you opened with **file_open**.

There are two functions that generate output within the testing environment. The **report_msg** function prints a user-defined string expression to the test execution report. The **pause** function stops test execution and displays a string expression in a message box on the screen.

For more information on any of the TSL built-in functions, see the TSL Reference Manual.

Comments

A number sign (#) indicates that all text from this point to the end of the line is a comment. Comments can appear within statements that extend beyond one line, or can stand alone on a line of test script. The TSL interpreter does not process comments. For example,

```
# Type the date
i=1
while (i<=31)# number of days in month
    type ("The date is January " & i++ & ", 1994");
```

Note that a number sign (#) that appears within a string constant is not considered a comment; for instance, a="#3".

Built-in Functions

TSL provides numerous built-in functions that perform a range of tasks. To call a built-in function from within a test script, use the following syntax:

function ([*parameters*]);

Most built-in functions return a value. This value can be assigned to a variable. For example,

```
x = int(12.42);
```

The **int** function returns the integer portion of a positive, real number. Here, x is equal to 12.

The return value of a built-in function can also become part of an expression. When a function returns the value 0, the value of the expression is considered false. When it returns any other value, it is considered true. For example,

```
while (getline address < "clients.doc")
    type (address "<kReturn>");
```

The **getline** function returns the value 1 if it succeeds, and 0 at the end of the file. Therefore, the *while* loop above continues until the end of the file is reached (the function returns the value 0).

For detailed information on each of the TSL functions, see the TSL Reference Manual.

User-Defined Functions

In addition to the built-in functions it offers, TSL allows you to design and implement your own functions in test scripts. A user-defined function has the following structure:

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

Class

The class of a function may be either public or static. If no class is explicitly declared, the function is assigned the default class public. A public function is available to all tests; a static function is available only to the test or compiled module within which the function was defined.

Parameters

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

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

out: A parameter which passes a value from inside the function.

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

A parameter designated as *out* or *inout* must be a variable name, not an expression. Only a variable can be assigned a value in a function call, not an expression. For example, consider a function defined in the following manner:

```
function my_func (out p) {... }
```

Proper usage of the function call is: my_func (var_1); Illegal usage of the function call is: my_func (arr[i]); my_func (a+b); because arr[i] and a+b are expressions.

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

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

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

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

Declarations

Variables used by a function must be declared. The declaration for such a variable can be within the function itself, or anywhere else within the test or module. For syntax, **see Variable Declarations,** in this chapter.

Return Statement

Any valid statement used within a TSL test script can be used within a function. In addition, the *return* statement is used exclusively in functions.

return [*expression*]**;**

This statement halts execution of the called function and 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 attached to the return statement, an empty string is returned.) For additional information on functions, refer to your *User's Guide.*

External Function Declarations

The extern function declaration is used to declare functions that are not part of TSL, but reside in external C libraries. For more information on using C functions stored in external dlls, refer to your *User's Guide*.

The extern declaration must appear before the function can be called. The syntax of the extern function declaration is:

extern *type function_name* **(** *param1, param2*,...**);**

The *type* refers to the return value of the function. 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*)

long (signed and unsigned)

Each parameter must include the following information:

[*mode*] *type* [*name*] [< *size* >]

mode	The *mode* can be *in*, *out*, or *inout*. The default is *in*. Note that these values must appear in lower case.
type	The *type* can be any of the values listed above.
name	An optional *name* can be assigned to the parameter to improve readability.
size	This information is required only for an *out* or *inout* parameter of type *string*. (See below.)

For example, to declare a function named set_clock that sets the time in a clock application, you write the following:

```
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 was successful.

Once the extern declaration is interpreted, you can call the set_clock function the same way you call a TSL built-in 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. It 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 127. There is no maximum size.

TSL identifies the function in your C code by its name only. You must pass the correct parameter information from TSL to the C function. TSL does not check parameters: if the information is incorrect, the operation fails.

In addition, your C function must adhere to the following conventions:

- Any parameter designated as a *string* in TSL must be associated with a parameter of type *char** in C.
- Any parameter of mode *out* or *inout* in TSL must be associated with a pointer in C. For instance, a parameter *out int* in TSL must be associated with a parameter *int** in the C function.
- For WinRunner the external function must observe the standard Pascal calling convention *export far Pascal*.

Find
?

For example, the following declaration in TSL:

```
extern int set_clock (string name, inout int time);
```

must appear as follows in C:

```
int _far _pascal _export [_loads] set_clock (
        char* name,
        int* time
)
```

Index

Find

?

Click a page

Find
?
Click a page
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Find
?
Click a page

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Find
?
Click a page
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Find
?
Click a page
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Find
?
Click a page

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Find
?
Click a page
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Find
?
Click a page
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Find
?
Click a page
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

WinRunner User’s Guide

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

Mercury Interactive Corporation
1325 Borregas Avenue
Sunnyvale, CA 94089
Tel. (408)822-5200 (800) TEST-911
Fax. (408)822-5300

WRUG5.0/01