WinRunner®
User's Guide
Version 6.0

Online Guide

Contents Summary

Books Online

Find

Find Again

Help

Top of Chapter

Back

PART III: CREATING TESTS

Books
Online

Find

Find
Again

Help

Top of
Chapter

Back

Books Online

Find

Find Again

Help

Top of Chapter

Back

Books Online

Find

Find Again

Help

Top of Chapter

Back

Table of Contents

Books
Online

Find

Find
Again

Help

Top of
Chapter

Back

PART II: UNDERSTANDING THE GUI MAP

Books
Online

Find

Find
Again

? **Help**

回
**Top of
Chapter**

⇐ **Back**

Books Online

Find

Find Again

? Help

Top of Chapter

⇐ Back

**Books
Online**

M Find

**Find
Again**

? Help

◄ ►

**回
Top of
Chapter**

← Back

PART III: CREATING TESTS

Books Online

Find

Find Again

Help

Top of Chapter

Back

Books Online

Find

Find Again

Help

Top of Chapter

Back

Books Online

Find

Find Again

Help

Top of Chapter

Back

Books Online

Find

Find Again

Help

Top of Chapter

Back

PART IV: PROGRAMMING WITH TSL

Books Online

Find

Find Again

Help

Top of Chapter

Back

Books Online

Find

Find Again

Help

Top of Chapter

Back

Books
Online

Find

Find
Again

Help

Top of
Chapter

Back

PART V: RUNNING TESTS

Books Online

Find

Find Again

Help

Top of Chapter

Back

Books
Online

Find

Find
Again

Help

**Top of
Chapter**

Back

Books Online

Find

Find Again

Help

Top of Chapter

Back

PART VII: CONFIGURING WINRUNNER

Books Online

Find

Find Again

Help

Top of Chapter

Back

Books Online

Find

Find Again

Help

Top of Chapter

Back

PART VIII: WORKING WITH TESTSUITE

Books Online

Find

Find Again

Help

Top of Chapter

Back

Books Online

Find

Find Again

Help

Top of Chapter

Back

Welcome to WinRunner

Welcome to WinRunner, Mercury Interactive's enterprise functional testing tool for Microsoft Windows applications. With WinRunner you can quickly create and run 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 your application's behavior.

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, analyze test results, and report defects detected in your application.

Part VI: Debugging Tests

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

Part VII: Configuring WinRunner

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

Part VIII:Working with TestSuite

Describes how WinRunner interacts with TestDirector and LoadRunner.

Books Online

Find

Find Again

Help

Top of Chapter

Back

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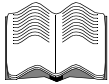
In addition to this guide, WinRunner comes with a complete set of documentation:

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

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

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

WebTest User's Guide teaches you how to use the WebTest add-in to test your Web site.

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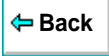
Online Resources

WinRunner includes the following online resources:

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

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

Books Online displays the complete documentation set in PDF format. Online books can be read and printed using Adobe Acrobat Reader 4.0, 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 Web 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 Web 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 Mercury Interactive's Customer Support Web site and home page, the e-mail address for requesting information, 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 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.

Books Online

Find

Find Again

Help

Top of Chapter

Back

Typographical Conventions

This book uses the following typographical conventions:

1, 2, 3	Bold numbers indicate steps in a procedure.
•	Bullets indicate options and features.
>	The greater than sign separates menu levels (for example, **File > Open**).
Bold	**Bold** text indicates function names.
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

Starting the Testing Process
Introduction

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

Recent advances 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, or 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 automate the testing process, from test development to execution. You create adaptable and reusable test scripts that challenge the functionality of your application. Prior to a software release, you can run 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 the Function Generator, which helps you 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 the object on the screen. Every time you perform an operation on the application being tested, a TSL statement describing the object selected and the action performed is generated in the test script.

As you record, WinRunner writes a unique description of each selected object to a GUI map. The GUI map consists of files maintained separately from your test scripts. If the user interface of your application changes, you have to update only the GUI map, instead of hundreds of tests. This allows you to easily reuse your Context Sensitive test scripts on future versions of your application.

To run a test, you simply play back the test script. WinRunner emulates a 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 being tested for objects matching these descriptions. It can locate objects in 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 run, 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 six main stages:

```
        Create Tests          Run Tests          Report Defects

Create                Debug Tests      View Results
GUI Map
```
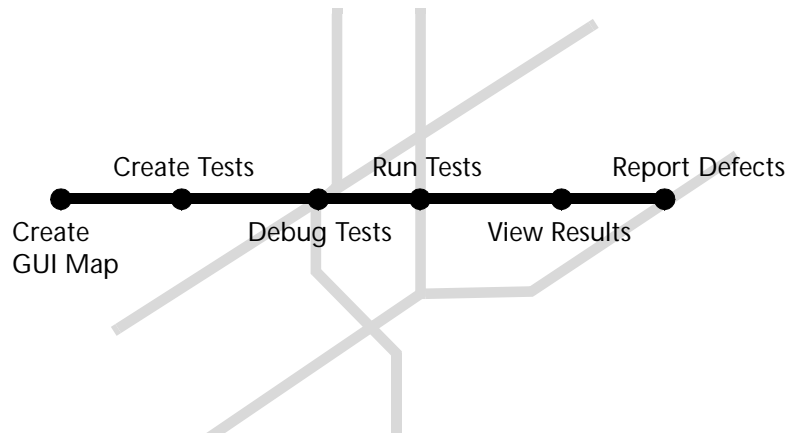
Create the GUI Map

The first stage is to create the GUI map so WinRunner can recognize the GUI objects in your application being tested. 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.
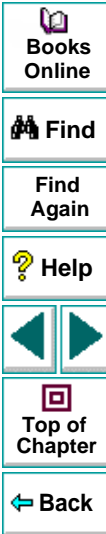
Create Tests

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 being tested. You can insert checkpoints that check GUI objects, bitmaps, and databases. During this process, WinRunner captures data and saves it as *expected results*—the expected response of the application being tested.

Debug Tests

You run tests in Debug mode to make sure they run smoothly. You can set breakpoints, monitor variables, and control how tests are run to identify and isolate defects. Test results are saved in the debug folder, which you can discard once you finished debugging the test.

Run Tests

You run tests in Verify mode to test your application. Each time WinRunner encounters a checkpoint in the test script, it compares the current data of the application being tested to the expected data captured earlier. If any mismatches are found, WinRunner captures them as *actual results*.

View Results

You determine the success or failure of the tests. Following each test run, WinRunner displays the results 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, 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.

Report Defects

If a test run fails due to a defect in the application being tested, 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.

Sample Application

Many examples in this book use the sample Flight Reservation application provided with WinRunner. Note that this application is Year 2000 compliant.

Starting the Sample Application

You can start this application by choosing **Start > Programs > WinRunner > Sample Applications** and then choosing the version of the flight application you want to open: Flight 1A or Flight 1B.

Multiple Versions of the Sample Application

The sample Flight Reservation application comes in two versions: Flight 1A and Flight 1B. Flight 1A is a fully working application, while Flight 1B has some "bugs" built into it. These versions are used together in the *WinRunner Tutorial* to simulate the development process, in which the performance of one version of an application is compared with that of another. You can use the examples in this guide with either Flight 1A or Flight 1B.

When WinRunner is installed with Visual Basic support, Visual Basic versions of Flight 1A and Flight 1B applications are installed in addition to the regular sample applications.

Books Online

Find

Find Again

Help

Top of Chapter

Back

Logging In

When you start the sample Flight Reservation application, the Login dialog box opens. You must log in to start the application. To log in, enter a name of at least four characters and password. The password is "Mercury" and is not case sensitive.

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 testing, 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, run tests, and report and track defects. You can also create reports and graphs to help review the progress of planning tests, running tests, and tracking defects before a software release.

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

Books Online

Find

Find Again

Help

Top of Chapter

Back

LoadRunner

LoadRunner is Mercury Interactive's testing tool for client/server applications. Using LoadRunner, you can emulate 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.

Books Online

Find

Find Again

Help

Top of Chapter

Back

Starting the Testing Process
WinRunner at a Glance

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

This chapter describes:

- **Starting WinRunner**
- **The Main WinRunner Window**
- **The Test Window**
- **Using WinRunner Commands**
- **Loading WinRunner Add-Ins**

Books Online

Find

Find Again

Help

Top of Chapter

Back

Starting WinRunner

To start WinRunner, either click the WinRunner icon in the WinRunner program group or click **Start > Programs > WinRunner > WinRunner.** After several seconds, the WinRunner window opens. Note 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 being tested.

The first time you start WinRunner, the **Welcome to WinRunner** window opens. You can choose to create a new test, open an existing test, or run the RapidTest Script wizard.
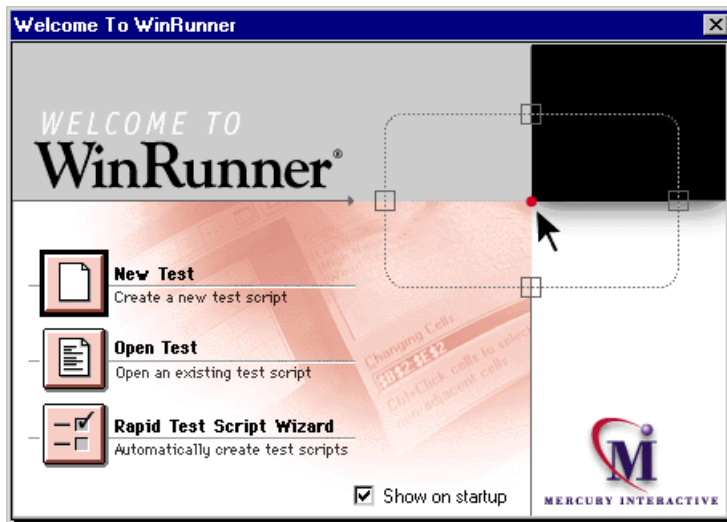
If you do not want this window to appear the next time you start WinRunner, clear the **Show at Startup** check box. To show the **Welcome to WinRunner** window upon startup from within WinRunner, select **Settings > General Options**, click the **Environment** tab, and select the **Show Welcome Screen** check box.
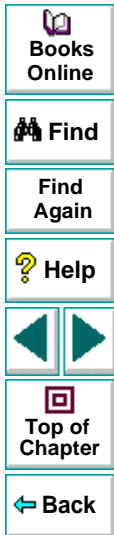
The Main WinRunner Window

The main WinRunner window contains the following key elements:

- *WinRunner title bar*

- *Menu bar,* with drop-down menus of WinRunner commands

- *Standard toolbar,* with buttons of commands commonly used when running a test

- *User toolbar,* with commands commonly used while creating a test

- *Status bar*, with information on the current command, the line number of the insertion point, and the name of the current results folder
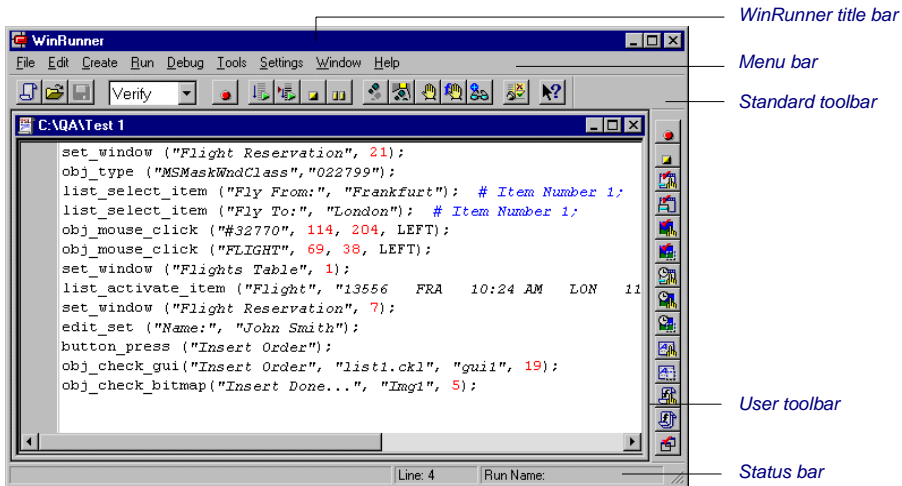
The Test Window

You create and run WinRunner tests in the test window. It contains the following key elements:

- *Test window title bar*, with the name of the open test

- *Test script*, with 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

Test window title bar

Execution arrow

Insertion point

Test script

```
set_window ("Flight Reservation", 21);
obj_type ("MSMaskWndClass","022799");
list_select_item ("Fly From:", "Frankfurt");  # Item Number 1;
list_select_item ("Fly To:", "London");  # Item Number 1;
obj_mouse_click ("#32770", 114, 204, LEFT);
obj_mouse_click ("FLIGHT", 69, 38, LEFT);
set_window ("Flights Table", 1);
list_activate_item ("Flight", "13556   FRA   10:24 AM   LON   11
set_window ("Flight Reservation", 7);
edit_set ("Name:", "John Smith");
button_press ("Insert Order");
obj_check_gui("Insert Order", "list1.ckl", "gui1", 19);
obj_check_bitmap("Insert Done...", "Img1", 5);
```

Books Online

Find

Find Again

Help

Top of Chapter

Back
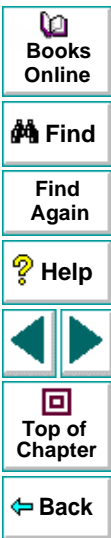
Using WinRunner Commands

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 *User toolbar*. You can customize the *User toolbar* with the commands you use most frequently.

Creating a Floating Toolbar

You can change a toolbar to a floating toolbar. This enables you to minimize WinRunner while maintaining access to the commands on a floating toolbar, so you can work freely with the application being tested.
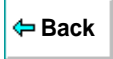
The Standard Toolbar

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 docked below the WinRunner menu bar. For more information about the Standard toolbar, see Chapter 27, **Running Tests**. The following buttons appear on the Standard toolbar:

Open *Record - Context Sensitive* *Run from Arrow* *Pause* *Step Into* *Break in Function* *Test Results*

New *Save* *Run Mode* *Run from Top* *Stop* *Step* *Toggle Breakpoint* *Add Watch* *Help*
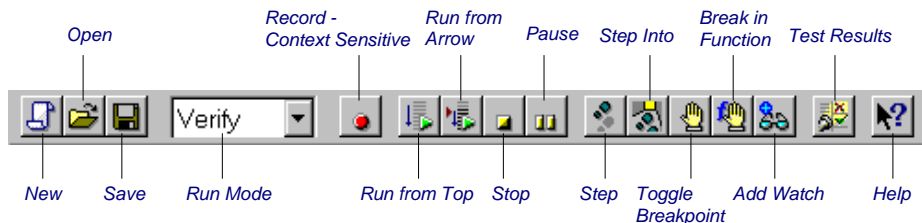
The User Toolbar

The User toolbar contains buttons for commands used when creating tests. By default, the User toolbar is hidden. To display the User toolbar, select it on the Window menu. When it is displayed, its default position is docked at the right edge of the WinRunner window. For information about creating tests, see Part III, Creating Tests.

The User toolbar is a customizable toolbar. You can add or remove buttons to facilitate access to commands commonly used for an application being tested . For information on customizing the User toolbar, see **Customizing the User Toolbar** on page 809. The following buttons appear by default on the User toolbar:

Record - Context Sensitive

Stop

GUI Checkpoint for Object/Window

GUI Checkpoint for Multiple Objects

Bitmap Checkpoint for Object/Window

Bitmap Checkpoint for Screen Area

Database Checkpoint

Synchronization Point for Object/Window Property

Synchronization Point for Object/Window Bitmap

Synchronization Point for Screen Area Bitmap

Get Text from Object/Window

Get Text from Screen Area

Insert Function for Object/Window

Insert Function from Function Generator

Books Online

Find

Find Again

Help

Top of Chapter

Back

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 being tested 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 833.

Loading WinRunner Add-Ins

If you installed add-ins such as support for Visual Basic, PowerBuilder, or ActiveX controls while installing WinRunner or afterward, you can specify which add-ins to load at the beginning of each WinRunner session.

When you start WinRunner, the **Add-In Manager** dialog box opens. It displays a list of all installed add-ins for WinRunner. You can select which add-ins to load for the current session of WinRunner. If you do not make a change within a certain amount of time, the window closes. The progress bar displays how much time is left before the window closes.
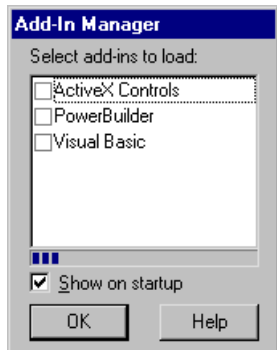
The first time WinRunner is started, by default, no add-ins are selected. At the beginning of each subsequent WinRunner session, your selection from the previous session is the default setting. Once you make a change to the list, the timer stops running, and you must click **OK** to close the dialog box.

You can determine whether to display the **Add-In Manager** dialog box and, if so, for how long using the **Display the Add-In Manager dialog** option in the Environment tab of the General Options dialog box. For information on working with the General Options dialog box, see Chapter 36, **Setting Global Testing Options**. You can also specify these options using the *-addins* and *-addins_select_timeout* command line options. For information on working with command line options, see Chapter 30, **Running Tests from the Command Line**.

Understanding the GUI Map

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:

- **How a Test Identifies GUI Objects**

- **Physical Descriptions**

- **Logical Names**

- **The GUI Map**

- **Setting the Window Context**

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 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 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 being tested. You can view the GUI map in order to gain a comprehensive picture of the objects in your application.

As the user interface of your application changes, you can continue to use tests you already developed. You simply add, delete, or edit object descriptions in the GUI map so WinRunner can continue to find the objects in your modified application.

Books Online

Find

Find Again

Help

Top of Chapter

Back

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*. This detailed description (the logical name and the physical description) ensures 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 might look like this:

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

Books Online

M Find

Find Again

? Help

Top of Chapter

← Back

WinRunner learns the actual description—the list of properties and their values—for each object 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 WinRunner 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

GUI Map

Books Online

Find

Find Again

Help

Top of Chapter

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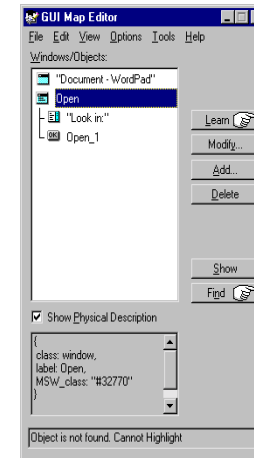
❶ *WinRunner reads the logical name in the test script and refers to the GUI map*

❷ *Matches the logical name with the physical description*

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

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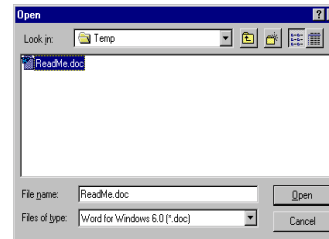
WinRunner identifies each GUI object in the application under test by its *physical description*: a list of physical properties and their assigned values. These property–value pairs appear in the following format in the GUI map:

{*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, push button, list, radio button, menu, etc. WinRunner identifies an object by learning its class property.

Each class has a 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 an object's physical description 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 65.

Books Online

Find

Find Again

Help

Top of Chapter

Back

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 text in the window's title bar; and for a list, the logical name is the text appearing next to or above the list.

For a static text object, the logical name is a combination 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.

Books Online

Find

Find Again

Help

Top of Chapter

Back

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.

Books Online

Find

Find Again

Help

Top of Chapter

Back

You can view the contents of the GUI map at any time by choosing Tools > GUI Map Editor. 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

GUI Map Editor

File   Edit   View   Options   Tools   Help

Windows/Objects:

"Document - WordPad"
Open
├ "Look in:"
└ Open_1

Learn
Modify...
Add...
Delete

Show
Find

Click to expand dialog box and display the physical description of the selected object or window

☑ Show Physical Description

```
{
class: window,
label: Open,
MSW_class: "#32770"
}
```

Object is not found. Cannot Highlight

Books Online

Find

Find Again

Help

Top of Chapter

Back
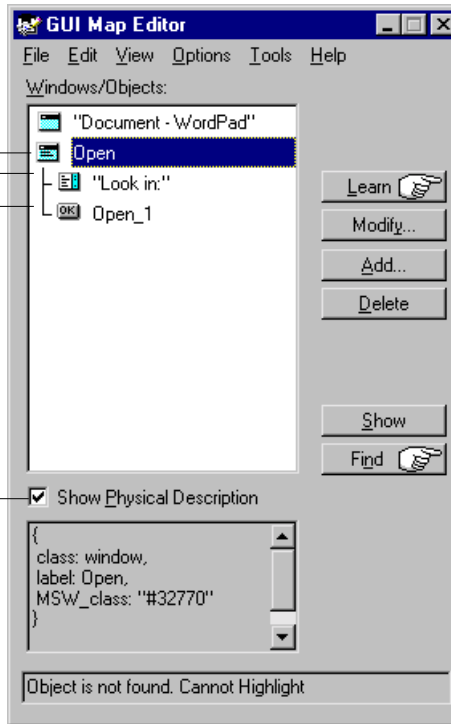
The GUI map enables you to easily keep up with changes made to the user interface of the application being tested. Instead of editing your entire suite of tests, you only have to update the object descriptions in the GUI map.
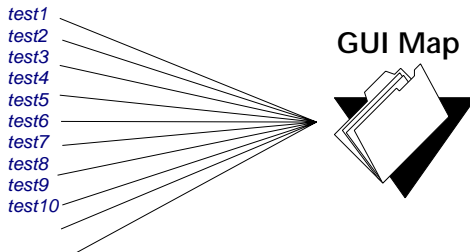
For example, suppose the Open button in the Open dialog box is changed to an OK button. You do not have to edit every test script that uses this Open button. Instead, you can modify the Open button's physical description in the GUI map, as shown in the example below. The value of the label property for the button is changed from Open to OK:

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

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

Books Online

M Find

Find Again

Help

Top of Chapter

Back

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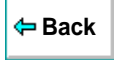
This chapter explains how to teach WinRunner the Graphical User Interface (GUI) of the application being tested and save the information for use during testing.

This chapter describes:

- **Learning the GUI with the RapidTest Script Wizard**
- **Learning the GUI by Recording**
- **Learning the GUI Using the GUI Map Editor**
- **Saving the GUI Map**
- **Loading the GUI Map File**

Books Online

Find

Find Again

Help

Top of Chapter

Back

About Creating the GUI Map

WinRunner can learn the GUI of your application in several ways. Usually, you use the RapidTest 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 individually relearn the GUI.

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 use in your application. This approach is fast and enables a beginning user to create test scripts immediately. This is an unsystematic method, however, and should not be used as a substitute for the RapidTest Script wizard if you plan 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. You should insert a **GUI_load** statement 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 39, **Initializing Special Configurations**. Alternatively, you can insert a **GUI_load** statement into individual tests, or use the GUI Map Editor to load GUI map files manually.
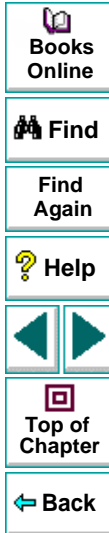
Learning the GUI with the RapidTest Script Wizard

The RapidTest Script wizard enables WinRunner to learn all windows and objects in your application being tested 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.

Books Online

M Find

Find Again

? Help

Note: The RapidTest Script wizard is not available for when working with the Terminal Emulator or WebTest add-ins. Therefore, if you are using these add-ins, do not complete this section or the following sections of this lesson. Instead, after you create and run tests in the other tutorial lessons, you must *save* your temporary GUI map file whenever you close WinRunner or close a test script you want to save. A temporary GUI map file contains the information that WinRunner learns about GUI objects in your application.

Top of Chapter

← Back

You can specify the name and location of your GUI map file. Your GUI map file must always have the .gui extension. When you create a new test or open an existing test in this tutorial, you must *load* the GUI map file. For additional information , see **Saving the GUI Map** on page 74 and **Loading the GUI Map File** on page 78.

To start the RapidTest Script wizard, either:

● Click **RapidTest Script Wizard** in the WinRunner Welcome screen when you start WinRunner.

● Choose **Create** > **RapidTest Script Wizard** at any time.

RapidTest Script Wizard

Welcome to the

Script Wizard !

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

☐ GUI
☐ Bitma
☐ U.I
☐ Template

To create your scripts, Script Wizard will:

● "Walk through" your application.

● Learn your application's Graphical User Interface (GUI).

Cancel | <<Back | Next>> | Help

For step-by-step information on using the RapidTest Script wizard, refer to the online *WinRunner Context Sensitive Help*.
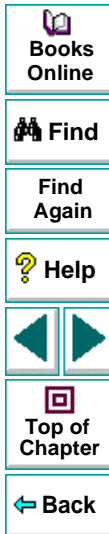
Learning the GUI by Recording

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 RapidTest Script wizard to learn the entire GUI of your application.

Books Online

Find

Find Again

? Help

Top of Chapter

Back

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 Choose **Tools** > **GUI Map Editor**. The GUI Map Editor opens.

**Books
Online**

Find

**Find
Again**

Help

**Top of
Chapter**

Back
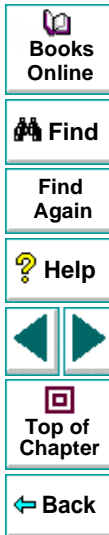
2 Click **Learn**. The mouse pointer becomes a pointing hand. (To cancel the operation, click the right mouse button.)
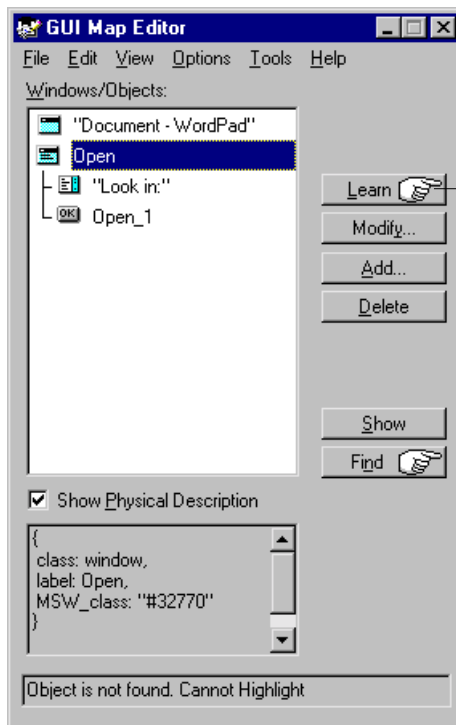
Learns the objects in a window.

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

Look in:	Temp

Open

ReadMe.doc

File name:	ReadMe.doc	Open
Files of type:	Word for Windows 6.0 (*.doc)	Cancel
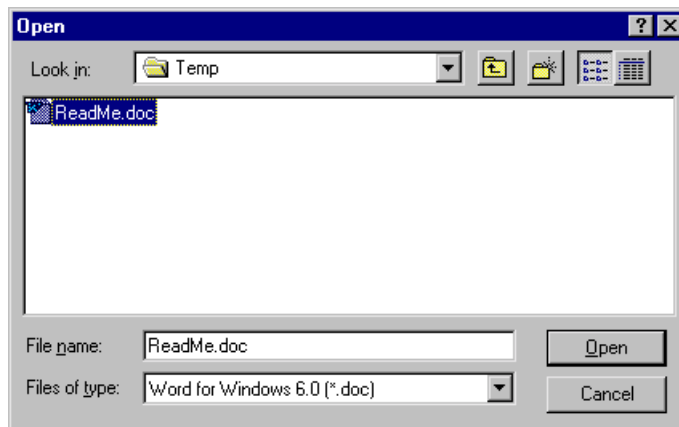
GUI information about the learned objects is placed in the active GUI map file. See **Loading the GUI Map File** on page 78 for more information.

Books Online

Find

Find Again

Help

Top of Chapter

Back

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 avoid overwriting valuable GUI information during a new recording session, save the temporary GUI map file in a permanent GUI map file.

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

1 Choose **Tools** > **GUI Map Editor**. The GUI Map Editor opens.

2 Choose **View** > **GUI Files**.

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

4 In the GUI Map Editor, choose **File** > **Save** to open the Save GUI File dialog box.
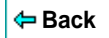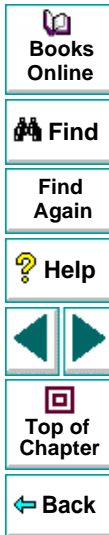
Note: If you add new windows from a loaded GUI map file to the temporary GUI map file, the New Windows dialog box opens. You are prompted to add the new windows to the loaded GUI map file or save them in a new GUI map file. For additional information, refer to the online *WinRunner Context Sensitive Help*.

Save GUI File

Save in: flight application

login1.gui

File name: login2

Save as type: GUI Files (*.gui)

Save

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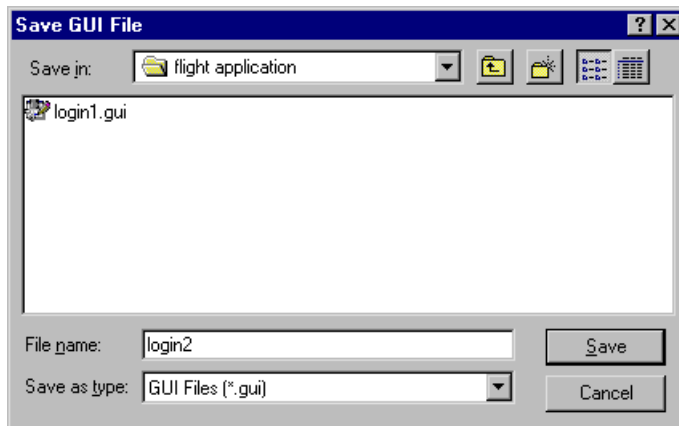
5 Click a folder. Type in a new file name or click an existing file.

6 Click **Save**. 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 Choose **View** > **GUI Files**.

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

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

Note: If you add new windows from a loaded GUI map file to the temporary GUI map file, the New Windows dialog box opens. You are prompted to add the new windows to the loaded GUI map file or save them in a new GUI map file. For additional information, refer to the online *WinRunner Context Sensitive Help*.

Books Online

Find

Find Again

Help

Top of Chapter

Back

The Save GUI File to TestDirector Project dialog box opens.
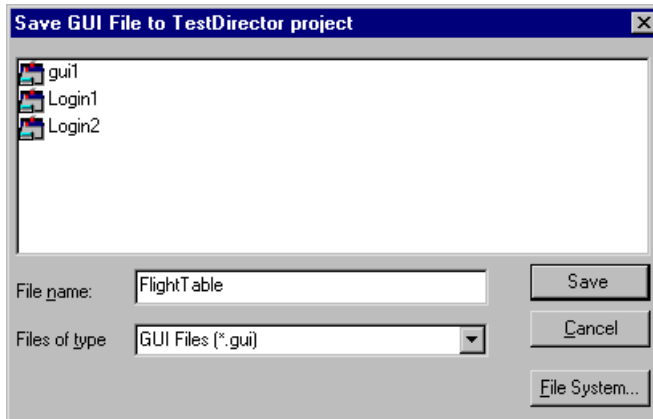
5 In the **File Name** text box, enter a name for the GUI map file. Use a descriptive name that will help you easily identify it later.

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

Note: You can only save GUI map files to a TestDirector database if you are working with TestDirector. For additional information, see Chapter 40, **Managing the Testing Process**.

Loading the GUI Map File

WinRunner uses GUI map files to locate objects in the application being tested. You must load the appropriate GUI map files before you run tests on your application being tested.

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

- using the **GUI_load** function
- 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" and a number preceding the file name.

Books Online

Find

Find Again

Help

Top of Chapter

Back

Loading GUI files Using the GUI_load Function

The **GUI_load** statement loads any GUI map file you specify. To load several files, use a separate statement for each. You can insert the **GUI_load** statement at the beginning of any test, but it is preferable to place it in your startup test. GUI map files are thereby loaded automatically each time you start WinRunner. For more information, see Chapter 39, **Initializing Special Configurations**.

To load a file using GUI_load:

1 Choose **File > Open** to open the test from which you want to load the file.

2 In the test script, 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 20, **Generating Functions**, for more information on the Function Generator.

3 Run the test to load the file. See Chapter 27, **Running Tests**, for more information.

Books Online

Find

Find Again

Help

Top of Chapter

Back

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 Choose **Tools** > **GUI Map Editor**. The GUI Map Editor opens.

2 Choose **View** > **GUI Files**.

3 Choose **File** > **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 **Open**. The GUI map file is added to the GUI file list. The letter "L" and a number preceding the file name 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 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.

Open GUI File from TestDirector project	✕

gui1
Login1
Login2

File name:	Login1	**Open**
Files of type	GUI Files (*.gui) ▾	Cancel
○ Load into the GUI Map ○ Open For Editing Only		File System...

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.

Books Online

Find

Find Again

Help

Top of Chapter

Back
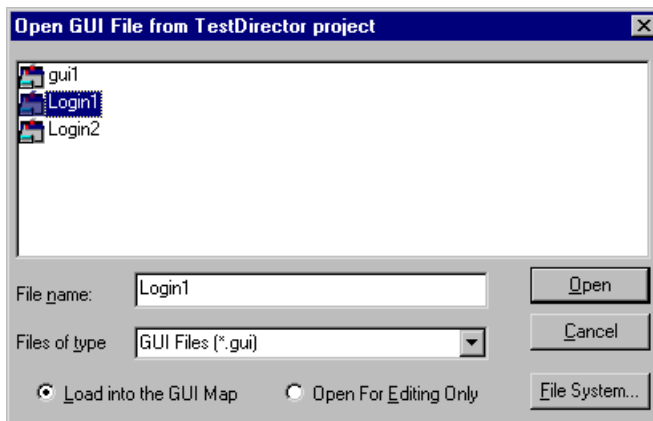
To load the GUI map file to open into the GUI Map Editor, make sure the **Load into the GUI Map** default setting is checked. Alternatively, if you only want to edit the GUI map file, click **Open for Editing Only**. For more information, see Chapter 5, **Editing the GUI Map**.

4 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.

Note: For more information on loading GUI map files from a TestDirector database, see Chapter 40, **Managing the Testing Process**.

This chapter explains how to extend the life of your tests by modifying descriptions of objects in the GUI map.

This chapter describes:

- **The Run Wizard**
- **The GUI Map Editor**
- **Modifying Logical Names and Physical Descriptions**
- **How WinRunner Handles Varying Window Labels**
- **Changing Regular Expressions in the Physical Description**
- **Copying and Moving Objects between Files**
- **Finding an Object in a GUI Map File**
- **Finding an Object in Multiple GUI Map Files**
- **Manually Adding an Object to a GUI Map File**
- **Deleting an Object from a GUI Map File**
- **Clearing a GUI Map File**
- **Filtering Displayed Objects**
- **Saving Changes to the GUI Map**

Books Online

Find

Find Again

Help

Top of Chapter

Back

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 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

● modify the logical names and physical descriptions of objects, add new descriptions, and remove obsolete descriptions.

● move or copy descriptions from one GUI map file to another.

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 the test run. During a test run, the Run wizard automatically happens 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 in your application.

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

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

☞ Click the Hand button in the wizard and point to the Network button. The Run wizard offers a solution.
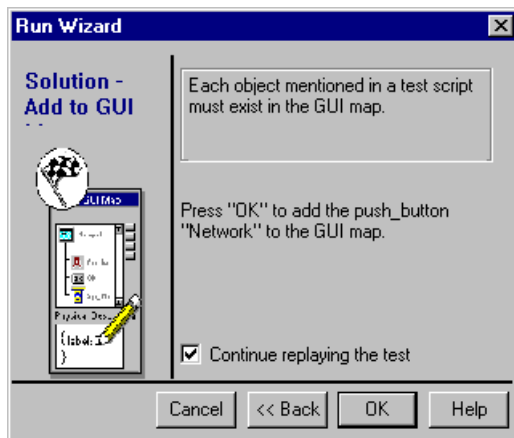
When you click OK, the Network object description is automatically added to the GUI map and WinRunner resumes the test The next time you run the test, WinRunner will be able to identify the Network button.

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, 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 **Tools** > **GUI Map Editor**.

Two views in the GUI Map Editor display the contents of either:

- the entire GUI map
- an individual GUI map file.
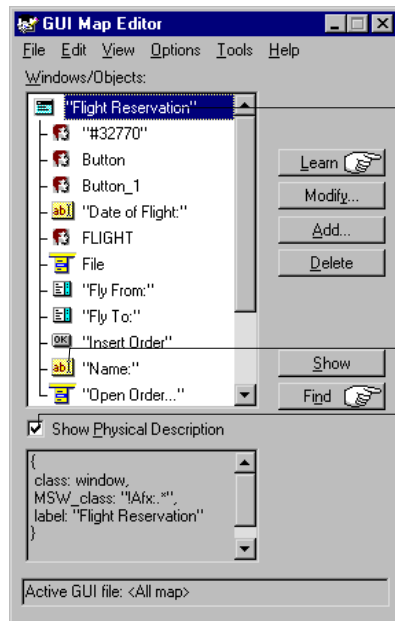
Displays all windows and objects in the GUI map.

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 enables you to easily copy or move descriptions between files. To view the contents of individual GUI map files, choose **View** > **GUI Files**.
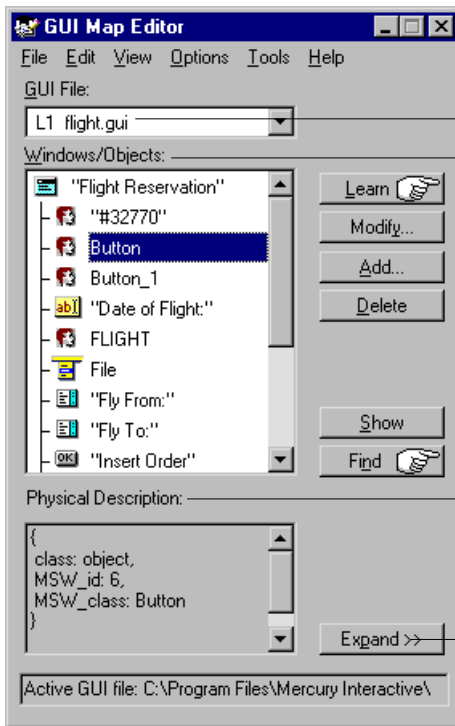
Lists the open GUI map files.

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 icon of the window in which they appear. When you double-click a window name or icon in the tree, you can view all the objects it contains. To concurrently view all the objects in the tree, choose **View** > **Expand Objects Tree**. To view windows only, choose **View** > **Collapse Objects 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.

Suppose the WordPad window is in your GUI map file. If you select **Show Physical Description** and click the WordPad window name or icon in the window list, the following physical description is displayed in the middle pane of the GUI Map Editor:

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

Note: If the value of a property contains any spaces or special characters, 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 assigned logical name 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 test scripts easier to read.

Changing the physical description is necessary when the property value of an object changes. For example, suppose 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".

Books Online

M Find

Find Again

? Help

Top of Chapter

← Back

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

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

2 Choose **View** > **GUI Files**.

3 If the appropriate GUI map file is not loaded, choose **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.
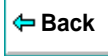
5 Select the name of the object or window to modify.

Click Modify.

Select a window or an object.

Books
Online

M Find

Find
Again

? Help

Top of
Chapter

← Back
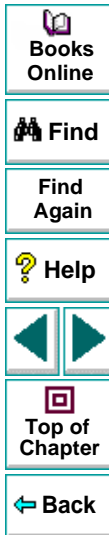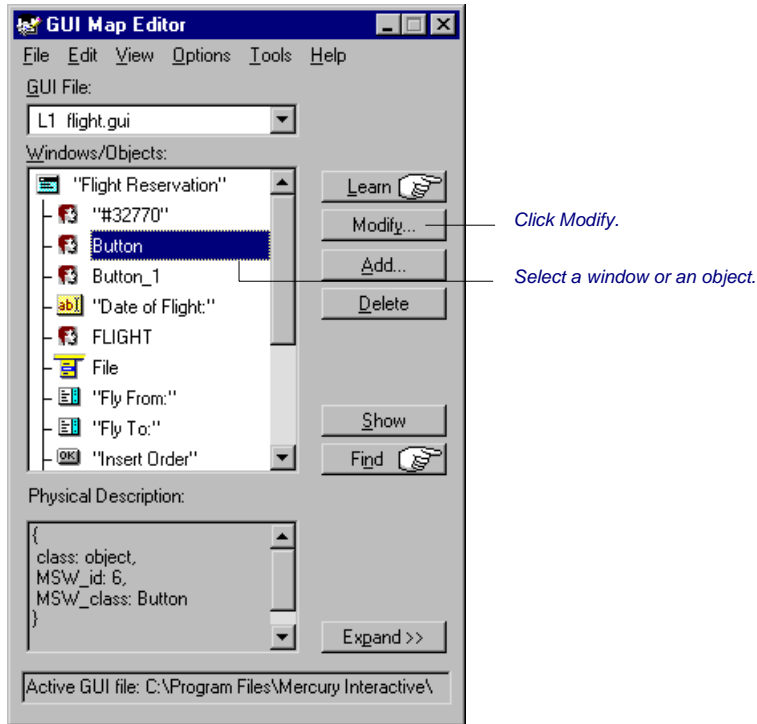
6 Click **Modify** to open the Modify dialog box.

```
Modify                                    ×
Logical Name:                          OK
Button
                                    Cancel
Physical Description:
                                      Help
{
class: object,
MSW_id: 6,
MSW_class: Button
}
```

Books Online

Find

Find Again

Help

Top of Chapter

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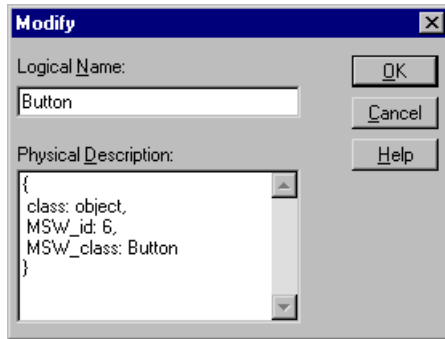
7 Edit the logical name or physical description as desired and click **OK**. The change appears immediately in the GUI map file.

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 as well as the application name in the title bar.

If WinRunner cannot recognize a window because its name changed after WinRunner learned it, the Run wizard opens and prompts you to identify the window in question. Once you identify the window, WinRunner realizes the window has a varying label, and it modifies the window's physical description accordingly.

Suppose you record a test on the main window of Microsoft Word in Windows 95. WinRunner learns the following physical description:

```
{
 class: window,
 label: "Microsoft Word - Document1",
 MSW_class: OpusApp
}
```

Books Online

M Find

Find Again

? Help

Top of Chapter

Back

Suppose you run your test when Document 2 is open in Microsoft Word. When WinRunner cannot find the window, the Run wizard opens:

You click the Hand button and click the appropriate Microsoft Word window, so that WinRunner will learn it. You are prompted to instruct WinRunner to update the window's description in the GUI map.
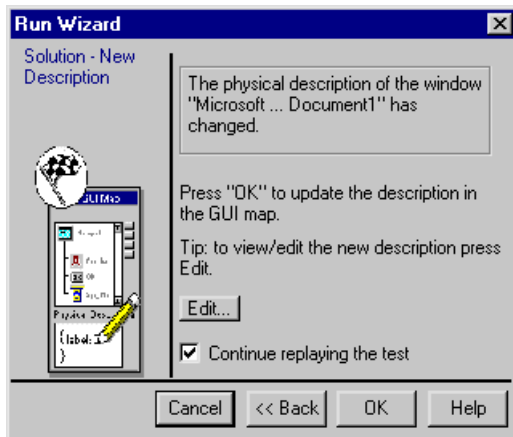
If you click Edit, you can see that WinRunner has modified the window's physical description to include regular expressions:

```
{
class: window,
label: "!Microsoft Word - Document.*",
MSW_class: OpusApp
}
```

(To continue running the test, you click OK.)

These regular expressions enable WinRunner to recognize the Microsoft Word window regardless of the name appearing after the dash in the window title. For additional information on regular expressions, see Chapter 26, **Using Regular Expressions**.

Changing Regular Expressions in the Physical Description

WinRunner uses two "hidden" properties in order to use a regular expression in an object's physical description. 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 object.

Adding a Regular Expression

You can add the regexp_label and the regexp_MSW_class properties 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 — ". For WinRunner to 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.

For more information about regular expressions, see Chapter 26, **Using Regular Expressions**.

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

2 Choose **View** > **GUI Files**.

3 Click **Expand** in the GUI Map Editor. The dialog box expands to display two GUI map files simultaneously.
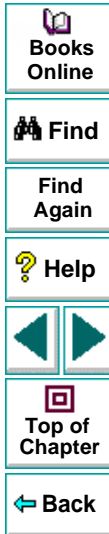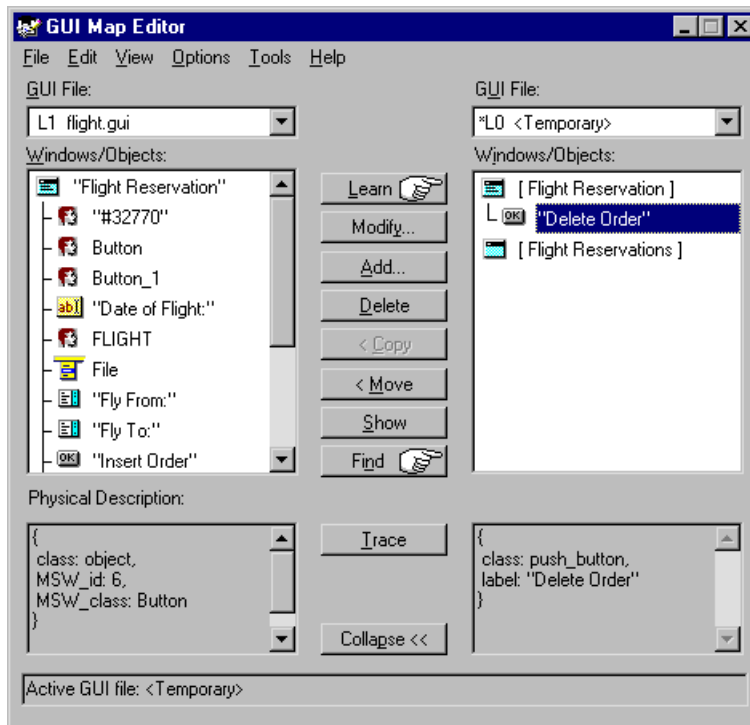
4 View a different GUI map file on each side of the dialog box by clicking the file names in the **GUI File** lists.

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

6 Click **Copy** or **Move**.

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

Books Online

Find

Find Again

Help

Top of Chapter

Back

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 **Tools** > **GUI Map Editor** to open the GUI Map Editor.

2 Choose **View** > **GUI Files**.

3 Choose **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.

Books Online

Find

Find Again

Help

Top of Chapter

Back

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 want WinRunner to 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 **Tools** > **GUI Map Editor** to open the GUI Map Editor.

2 Choose **View** > **GUI Files**.

3 Click **File > Open** to open the GUI map files in which the object description might appear.

For each file, choose **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 by displaying the GUI map file in the GUI File box.

5 Select the object in the **Windows/Objects** field.

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

7 Click **Trace**. 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.

Books Online

Find

Find Again

Help

Top of Chapter

Back

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 **Tools** > **GUI Map Editor** to open the GUI Map Editor.

2 Choose **View** > **GUI Files**.

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

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

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

```
Add                                    ×
Window Name:                       OK
Flight Reservation
                                 Cancel
Logical Name:
Delete Order                      Help
Physical Description:
{
 class: push_button,
 label: "Delete Order"
}
```
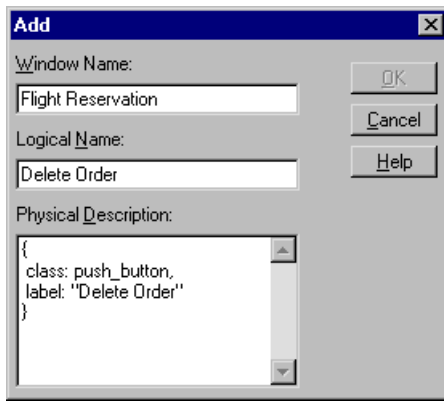
6 Edit the appropriate fields and click **OK**. The object is added to the GUI map file.

Books Online

Find

Find Again

Help

Top of Chapter

Back
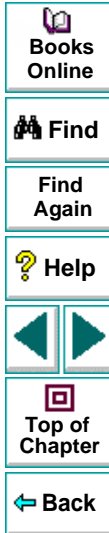
Deleting an Object from a GUI Map File

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 **Tools** > **GUI Map Editor** to open the GUI Map Editor.

2 Choose **View** > **GUI Files**.

3 Choose **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 and/or Control key to make your selection.

5 Click **Delete**.

6 Choose **File** > **Save** to save the changes to the GUI map file.

To delete all objects in a window:

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

2 Choose **View** > **GUI Files**.

3 Choose **File** > **Open** in the GUI Map Editor to open the appropriate GUI map file.

4 Choose **Edit** > **Clear All**.

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 **Tools** > **GUI Map Editor** to open the GUI Map Editor.

2 Choose **View** > **GUI Files**.

3 Open the appropriate GUI map file.

4 Display the GUI map file at the top of the GUI File list.

5 Choose **Edit** > **Clear All**.

Books Online

Find

Find Again

Help

Top of Chapter

Back

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 (e.g. "Open")
  or substring (e.g. "Op").

- *Physical description* displays only objects matching the specified physical
  description. Use any substring belonging to the physical description. (For
  example, specifying "w" filters out all objects containing a "w" in their physical
  description.)

- *Class* displays only objects of the specified class, such as all the push buttons.

Books Online

M Find

Find Again

? Help

Top of Chapter

Back

To apply a filter:

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

2 Choose **Options** > **Filters** to open the Filters dialog box.
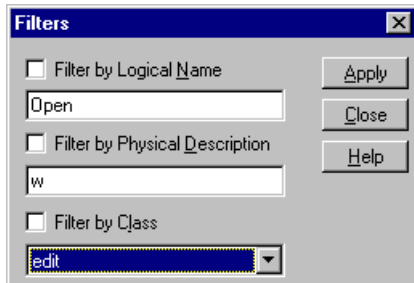
3 Select the type of filter you want by selecting a check box and entering the appropriate information.

4 Click **Apply**. The GUI Map Editor displays objects according to the filter applied.

**Books
Online**

Find

**Find
Again**

Help

**Top of
Chapter**

Back

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:

- Choose **File** > **Save** in the GUI Map Editor to save changes in the appropriate GUI map file.

- Choose **File** > **Save As** to save the changes in a new GUI map file.

Books Online

M Find

Find Again

❓ Help

Top of Chapter

⬅ Back

Understanding the GUI Map
Configuring the GUI Map

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

This chapter describes:

- **Viewing GUI Object Properties**
- **Understanding the Default GUI Map Configuration**
- **Mapping a Custom Object to a Standard Class**
- **Configuring a Standard or Custom Class**
- **Creating a Permanent GUI Map Configuration**
- **Deleting a Custom Class**
- **The Class Property**
- **All Properties**
- **Default Properties Learned**

Books Online

Find

Find Again

Help

Top of Chapter

Back
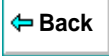
About Configuring the GUI Map

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

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

For example, a standard push button is defined by 26 properties, such as MSW_class, label, text, nchildren, x, y, height, class, focused, enabled. In most cases, however, WinRunner needs only the *class* and *label* properties to create a unique identification for the push button.

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

If a custom object is similar to a standard object, you can map it to one of the standard classes. You can also configure the properties WinRunner uses to identify a custom object during Context Sensitive testing. The mapping and the configuration you set are valid only for the current WinRunner session. To make the mapping and the configuration permanent, you must add configuration statements to your startup test script. Each time you start WinRunner, the startup test activates this configuration.

Books Online

Find

Find Again

Help

Note: If your application contains owner-drawn custom buttons, you can map them all to one of the standard button classes instead of mapping each button separately. You do this: by either choosing a standard button class in the Record Owner-Drawn Buttons box in the Record tab in the General Options dialog box; or, setting the rec_owner_drawn testing option with the **setvar** function from within a test script. For more information, see Chapter 37, **Setting Testing Options from a Test Script**.

Top of Chapter

Back

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

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 Choose **Tools** > **GUI Spy** to open the GUI Spy dialog box.

Books Online
M Find
Find Again
? Help
◀ ▶
回 Top of Chapter
⇦ Back

GUI Spy

Click the Spy button to spy on properties.

Window Name:

Object Name:

Description:

Spy 👉

Spy on
- ● Objects
- ○ Windows

Show in description
- ● Recorded properties
- ○ All properties

Help

Close

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 Description pane.

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 close the dialog box.

Understanding the Default GUI Map Configuration

For each class, WinRunner learns a set of default properties. Each default property is classified "obligatory" or "optional". (For a list of the default properties, see **All Properties** on page 138.)

- An *obligatory* property is always learned (if it exists).

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

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

{class:push_button, label:"OK"}

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

- A *location* selector uses the spatial position of objects.

Books Online

Find

Find Again

Help

Top of Chapter

Back

● 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. You should use this selector if the location of objects with the same description may change within a window. See **Configuring a Standard or Custom Class** on page 124 for more information.
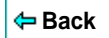
Mapping a Custom Object to a Standard Class

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

Using the GUI Map Configuration dialog box, you can teach WinRunner a custom object and map it to a standard class. For example, if your application has a custom button that WinRunner cannot identify, clicking this button is recorded as **obj_mouse_click**. You can teach WinRunner the "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 Choose **Tools** > **GUI Map Configuration** to open the GUI Map Configuration dialog box.
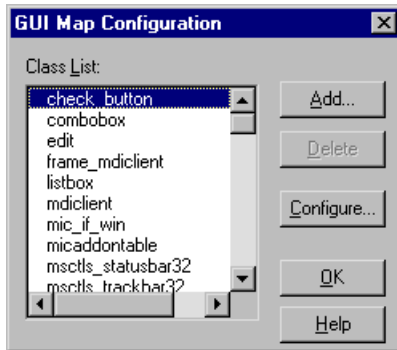
The Class List displays all standard and custom classes identified by WinRunner.

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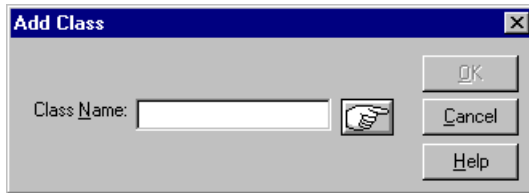
3 Click the pointing hand and then click the object whose class you want to add. The name of the custom object appears in the Class Name box. Note that this name is the value of the object's MSW_class property.

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

Books Online

Find

Find Again

Help

Top of Chapter

Back
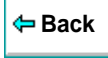
5 Click **Configure** to open the Configure Class dialog box.
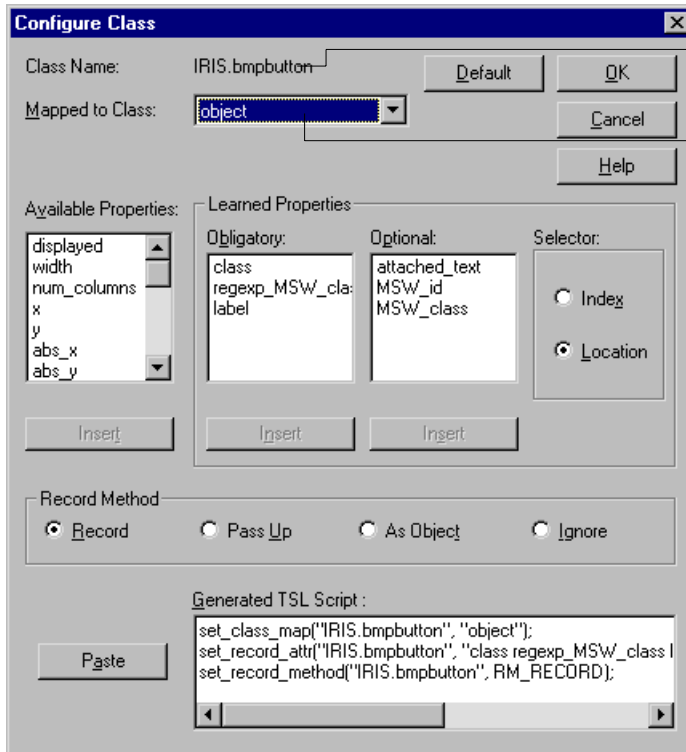
The custom class you are mapping

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 124.

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 132 for more information.

Books Online

M Find

Find Again

? Help

Top of Chapter

⇐ Back

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 recording method

To configure a standard or custom class:

1 Choose **Tools** > **GUI Map Configuration** to open the GUI Map Configuration dialog box.
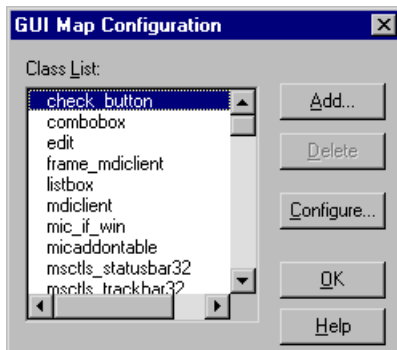
The Class List contains all standard classes, as well as any custom classes you add.
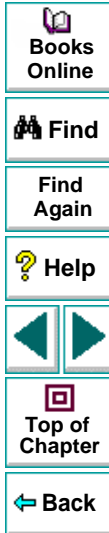
2 Click the class you want to configure and click **Configure**. The Configure Class dialog box opens.

Class you want to configure

Selector for the class

Obligatory and Optional properties learned for the class

All available properties

Record method for the class

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

3 Modify the learned properties, the selector, or the recording method as desired. See **Configuring Learned Properties** on page 126, **Configuring the Selector** on page 130, and **Configuring the Recording Method** on page 130 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 132 for more information.

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

Configuring Learned Properties

The Learned Properties area of the Configure Class dialog box allows you to configure which properties are recorded and learned for a class. You do this by 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 always learned (provided that they are valid for the specific object).
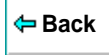
- The Optional list contains properties used only if the obligatory properties do not provide a unique identification for an object. WinRunner selects the minimum number of properties needed to identify the object, beginning with the first property in the list.

- The Available Properties list contains all remaining properties 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 appearing in the Class Name field.

  To modify the property configuration:

1 Click a property to move from any of the lists. Then click **Insert** 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.
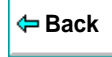
Note that not all properties apply to all classes. The following table lists each property and the classes to which it can be applied.

Property	Classes
abs_x	All classes
abs_y	All classes
active	All classes
attached_text	combobox, edit, listbox, scrollbar
class	All classes
displayed	All classes
enabled	All classes
focused	All classes
handle	All classes
height	All classes
label	check_button, push_button, radio_button, static_text, window
maximizable	calendar, window
minimizable	calendar, window
MSW_class	All classes

Books Online

Find

Find Again

Help

Top of Chapter

Back

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

Books Online

Find

Find Again

Help

Top of Chapter

Back

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 117.

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

Configuring the Recording Method

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

- *Record* instructs WinRunner to record all operations performed on a GUI object. This is the default record method for all classes. (The only exception is the static class (static text), for which the default is *Pass Up.*)

- *Pass Up* instructs WinRunner to record an operation performed on this class as an operation performed on the element containing the object. Usually this element is a window, and the operation is recorded as **win_mouse_click**.

- As Object instructs WinRunner to record all operations performed on a GUI object as though its class were "object" class.

- *Ignore* instructs WinRunner to disregard all operations performed on the class.

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

Creating a Permanent GUI Map Configuration

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

To create a permanent GUI map configuration for a class:

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

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

Books Online

M Find

Find Again

? Help

Top of Chapter

⬅ Back

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

Configure Class				☒
Class Name:	check_button		Default	OK
Mapped to Class:	check_button ▼			Cancel
				Help

Available Properties:

displayed
width
num_columns
x
y
abs_x
MSW_class ▼

Learned Properties

Obligatory:

class
label

Optional:

MSW_id

Selector:

○ Index

◉ Location

Insert   Insert   Insert

Record Method

◉ Record   ○ Pass Up   ○ As Object   ○ Ignore

Generated TSL Script :

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

Paste

*TSL statements
describing the GUI map
configuration*
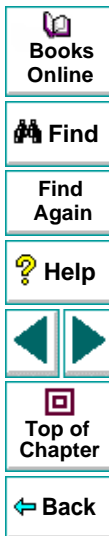
4 Paste the TSL statements into a startup test using the **Paste** button.

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

[WrEnv]
XR_TSL_INIT = GS:\tests\my_init

You would open the my_init test in the WinRunner window and paste in the generated TSL lines.
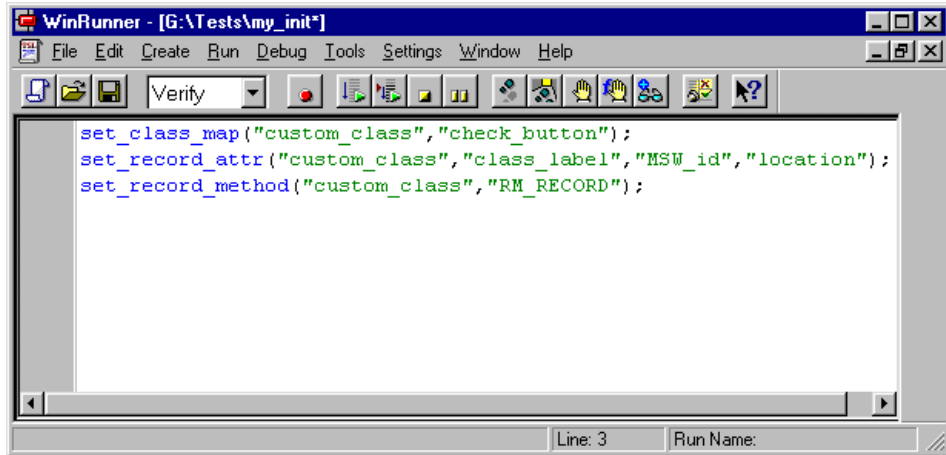
```
set_class_map("custom_class","check_button");
set_record_attr("custom_class","class_label","MSW_id","location");
set_record_method("custom_class","RM_RECORD");
```

For more information on startup tests, see Chapter 39, **Initializing Special Configurations**. For more information on the TSL functions defining a custom GUI map configuration (**set_class_map**, **set_record_attr**, and **set_record_method**), refer to the *TSL Online Reference*.
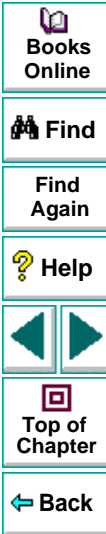
Deleting a Custom Class

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

To delete a custom class:

1 Choose **Tools** > **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**.

Books Online

Find

Find Again

? Help

Top of Chapter

Back

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
calendar	A standard calendar object that belongs to the *CDateTimeCtrl* or *CMonthCalCtrl* MSW_class.
check_button	A check box
edit	An edit field
frame_mdiclient	Enables WinRunner to treat a window as an mdiclient object.
list	A list box. This can be a regular list or a combo box.
menu_item	A menu item
mdiclient	An mdiclient object
mic_if_win	Enables WinRunner to defer all record and run operations on any object within this window to the mic_if library. Refer to the *WinRunner Customization Guide* for more information.
object	Any object not included in one of the classes described in this table
push_button	A push (command) button

Class	Description
radio_button	A radio (option) button
scroll	A scroll bar or slider
spin	A spin object
static_text	Display-only text not part of any GUI object
status bar	A status bar on a window
tab	A tab item
toolbar	A toolbar object
window	Any application window, dialog box, or form, including MDI windows

Books Online

M Find

Find Again

? Help

Top of Chapter

Back

All Properties

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

Note for XRunner users: You cannot use GUI maps 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 14, **Checking Bitmaps**.

Books Online

Find

Find Again

Help

Top of Chapter

Back

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 136.
class_index	An index number that identifies an object, relative to the position of other objects from the same class in the window (Java add-in only).
count	The number of menu items contained in a menu.
displayed	A Boolean value indicating whether the object is displayed: 1 if visible on screen, 0 if not.
enabled	A Boolean value indicating whether the object can be selected or activated: 1 if enabled, 0 if not.
focused	A Boolean value indicating whether keyboard input will be directed to this object: 1 if object has keyboard focus, 0 if not.
height	Height of object in pixels.
html_url	A URL (WebTest only).

Books Online

Find

Find Again

Help

Top of Chapter

Back

Property	Description
label	The text that appears on the object, such as a button label.
maximizable	A Boolean value indicating whether a window can be maximized: 1 if the window can be maximized, 0 if not.
minimizable	A Boolean value indicating whether a window can be minimized: 1 if the window can be minimized, 0 if not.
module_name	The name of an executable file which created the specified window.
nchildren	The number of children the object has: the total number of descendants of the object.
NSTBTitle	The title of a toolbar in a browser (WebTest only).
NSTitle	The title of a browser (WebTest only).
num_columns	A table object in Terminal Emulator applications only.
num_rows	A table object in Terminal Emulator applications only.
parent	The logical name of the parent of the object.
part_value	The name of a radio button or a check box in a group (WebTest only).
position	The position (top to bottom) of a menu item within the menu (the first item is at position 0).

Property	Description
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.

Books Online

M Find

Find Again

? Help

Top of Chapter

← Back

Semi-Portable Properties

Property	Description
handle	A run-time pointer to the object: the HWND handle.
TOOLKIT_class	The value of the specified toolkit class. The value of this property is the same as the value of the MSW_class in Windows, or the X_class in Motif.

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_name	A concatenation of the DataWindow and column names. For edit field objects in WinRunner with PowerBuilder add-in support, indicates the name of the column.
owner	(For windows), the application (executable) name to which the window belongs.

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_MSWclass	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.)

Books Online

M Find

Find Again

? Help

Top of Chapter

← Back

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

Books Online

Find

Find Again

Help

◀ ▶

Top of Chapter

⟵ **Back**

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: 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:

- **Defining a Virtual Object**
- **Understanding a Virtual Object's Physical Description**

About Learning Virtual Objects

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

For example, suppose you record a test on the Windows 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);
```

Books Online

M Find

Find Again

? Help

Top of Chapter

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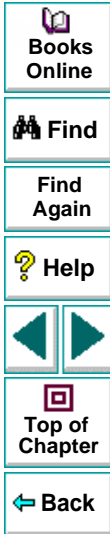
This test script is difficult to understand. If, instead, you define the calculator buttons as virtual objects and associate them with the push button class, WinRunner records a script similar to the following:

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

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

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

Defining a Virtual Object

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

To define a virtual object using the Virtual Object wizard:

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

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

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

Click **Next**.

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

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

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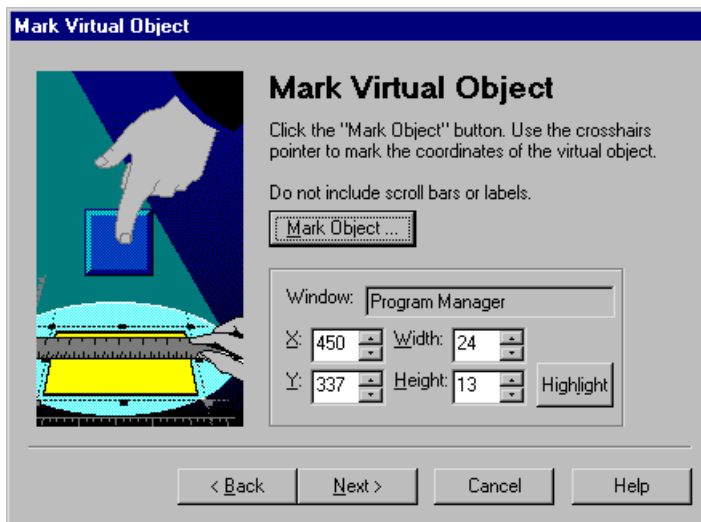
If the object marked is visible on the screen, you can click the Highlight button to view it. Click **Next**.

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.

Virtual Object Wizard

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,
```

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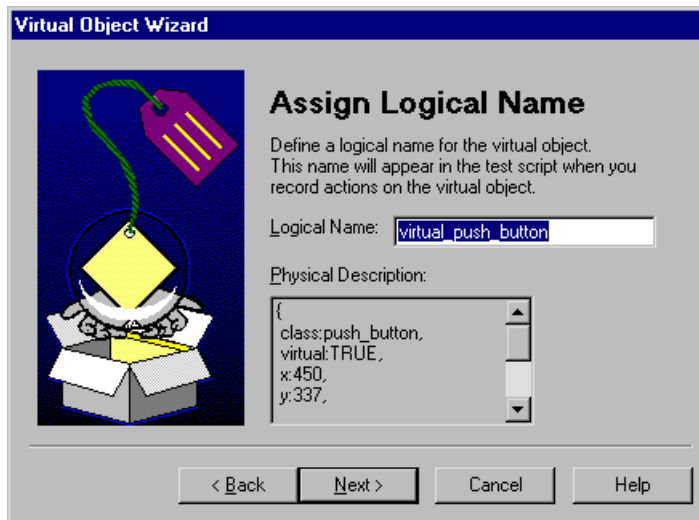
You can accept the wizard's suggestion or type in a different name. WinRunner checks that there are no other objects in the GUI map with the same name before confirming your choice. Click **Next**.

5 Finish learning the virtual object:

- If you want to learn another virtual object, click **Yes**. Click **Next**.

- To close the wizard, click **Finish**.

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

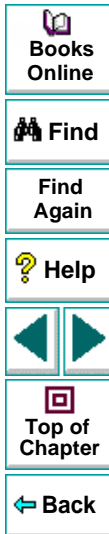When you create a virtual object, WinRunner adds its physical description to the GUI map. The physical description of a virtual object does not contain the *label* property found in the physical description of "real" GUI objects. Instead it contains a special property, *virtual*. Its function is to identify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

Books Online

Find

Find Again

Help

Top of Chapter

Back

Creating Tests

Creating Tests
Creating Tests

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

This chapter describes:

- **The WinRunner Test Window**
- **Context Sensitive Recording**
- **Analog Recording**
- **Checkpoints**
- **Data-Driven Tests**
- **Synchronization Points**
- **Planning a Test**
- **Documenting Test Information**
- **Recording a Test**
- **Activating Test Creation Commands Using Softkeys**
- **Programming a Test**
- **Editing a Test**
- **Managing Test Files**

Books Online

Find

Find Again

Help

Top of Chapter

Back

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 add GUI, bitmap, text, and database checkpoints, as well as synchronization points to your test script. Checkpoints enable 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.

You can create a data-driven tests, which are tests driven by data stored in an internal table.

Books Online

Find

Find Again

Help

Top of Chapter

Back

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 project.

Books Online

Find

Find Again

Help

Top of Chapter

Back
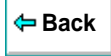
The WinRunner Test Window

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


```
set_window ("Flight Reservation", 5);
obj_mouse_click ("Button", 17, 7, LEFT);
obj_type ("MSMaskWndClass","121299");
list_select_item ("Fly From:", "Frankfurt"); # Item Number 1;
list_select_item ("Fly To:", "Los Angeles"); # Item Number 2;
obj_mouse_click ("FLIGHT", 35, 37, LEFT);
set_window ("Flights Table", 2);
list_select_item ("Flight", "20331 FRA  09:12 AM  LAX 04:23 PM  AA $112.20"); #
button_press ("OK");
set_window ("Flight Reservation", 4);
edit_set ("Name:", "John Smith");
button_set ("First", ON);
button_press ("Insert Order");
```

Test window title bar
Execution arrow
Insertion point
Test script

Books Online
Find
Find Again
Help
Top of Chapter
Back

Context Sensitive Recording

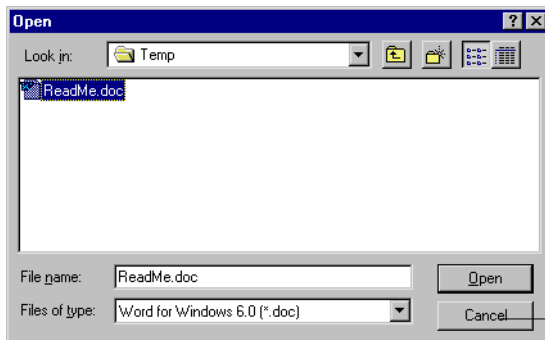*Context Sensitive* mode records the operations you perform on your application in terms of its GUI objects. As you record, WinRunner identifies each GUI object you click (such as a window, button, or list), and the type of operation performed (such as drag, click, or select).

For example, if you click the Open button in an Open dialog box, WinRunner records the following:
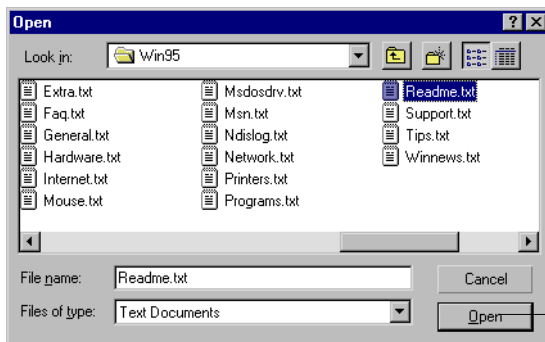
button_press ("Open");

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.

In version 1, the Open button is above the Cancel button.

In version 2, the Open button is below the Cancel button.

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.

Books Online

Find

Find Again

Help

Top of Chapter

Back
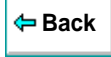
Analog Recording

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 for XRunner users: You cannot run test scripts in WinRunner that were recorded in XRunner in Analog mode. 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 14, **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 four 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 14, **Checking Bitmaps**, for more information.

● Text checkpoints read text in GUI objects and in bitmaps and enable you to verify their contents. See Chapter 15, **Checking Text**, for more information.

● Database checkpoints check the contents and the number of rows and columns of a result set, which is based on a query you create on your database. See Chapter 13, **Checking Databases**, for more information.

Books Online

Find

Find Again

Help

Top of Chapter

Back

Data-Driven Tests

When you test your application, you may want to check how it performs the same operations with multiple sets of data. You can 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*. You can perform these operations manually, or you can use the DataDriver Wizard to parameterize your test and store the data in a data table. For additional information, see Chapter 16, **Creating Data-Driven Tests**.

Synchronization Points

Synchronization points enable 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 17, **Synchronizing the Test Run**, for more information.

Books Online

Find

Find Again

Help

Top of Chapter

Back

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 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.

Books Online

Find

Find Again

Help

Top of Chapter

Back

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 23, **Creating Compiled Modules**, and Chapter 21, **Calling Tests**.

Books Online

M Find

Find Again

? Help

Top of Chapter

Back

To document test information:

1 Choose **File** > **Test Properties** to open the **Test Properties** dialog box.
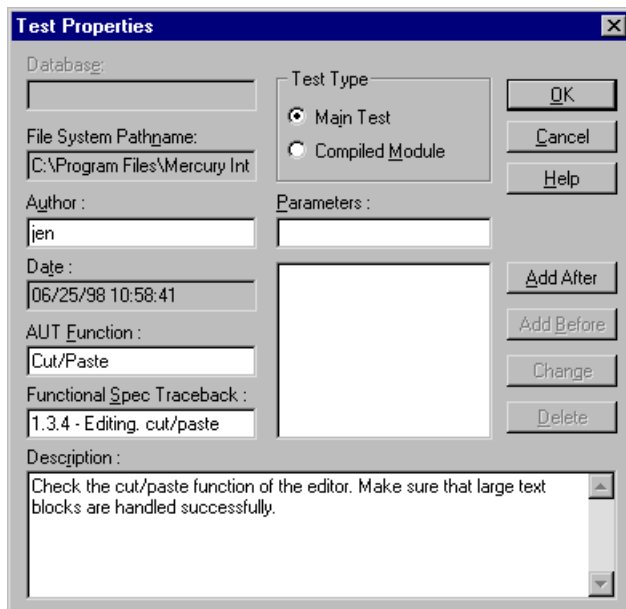
2 Add information about the test. Note that the Test Properties dialog box automatically displays the current date and time.

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 you record a test, you can minimize WinRunner and turn the User toolbar into a floating toolbar. This enables you to record on a full screen of your application, while maintaining access to important menu commands. To minimize WinRunner and work from the floating User toolbar: undock the User toolbar from the WinRunner window, start recording, and minimize WinRunner. The User toolbar stays on top of all other applications. Note that you can customize the User toolbar with the menu commands you use most frequently when creating a test. For additional information, see Chapter 34, **Customizing WinRunner's User Interface**.

- 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.
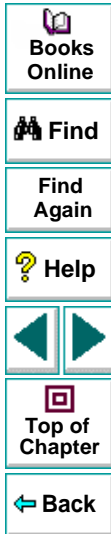
- When recording in Analog mode, avoid typing ahead. For example, when you want to open a window, wait until it is completely redrawn before continuing. In addition, avoid holding down a mouse button when this results in a repeated action (for example, using the scroll bar to move the screen display). 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 36, **Setting Global Testing Options**, and Chapter 37, **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 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 information, refer to the *TSL Online Reference*.

- If the test folder or the test script file is marked as read-only in the file system, you cannot perform any WinRunner operations which change the test script or the expected results folder.

To record a test:

1 Choose either **Create** > **Record–Context Sensitive** or
Create > **Record–Analog** or click the **Record–Context Sensitive** button.

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 User toolbar or from the Create menu: GUI Checkpoint, Bitmap Checkpoint, Database Checkpoint, or Synchronization Point.

3 To stop recording, click **Create** > **Stop Recording** or click **Stop**.
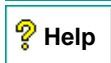
Activating Test Creation Commands Using Softkeys

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 34, **Customizing WinRunner's User Interface**.

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

Command	Default Softkey Combination	Function
RECORD	F2	Starts test recording. While recording, this softkey toggles between the Context Sensitive and Analog modes.
CHECK GUI FOR SINGLE PROPERTY	Alt Right + F12	Checks a single property of a GUI object.
CHECK GUI FOR OBJECT/WINDOW	Ctrl Right + F12	Creates a GUI checkpoint for an object or a window.
CHECK GUI FOR MULTIPLE OBJECTS	F12	Opens the Create GUI Checkpoint dialog box.
CHECK BITMAP OF OBJECT/WINDOW	Ctrl Left + F12	Captures an object or a window bitmap.

Books Online

Find

Find Again

Help

Top of Chapter

Back

Command	Default Softkey Combination	Function
CHECK BITMAP OF SCREEN AREA	Alt Left + F12	Captures an area bitmap.
CHECK DATABASE (DEFAULT)	Ctrl Right + F9	Creates a check on the entire contents of a database.
CHECK DATABASE (CUSTOM)	Alt Right + F9	Checks the number of columns, rows and specified information of a database.
SYNCHRONIZE OBJECT/WINDOW PROPERTY	Ctrl Right + F10	Instructs WinRunner to wait for a property of an object or a window to have an expected value.
SYNCHRONIZE BITMAP OF OBJECT/WINDOW	Ctrl Left + F11	Instructs WinRunner to wait for a specific object or window bitmap to appear.
SYNCHRONIZE BITMAP OF SCREEN AREA	Alt Left + F11	Instructs WinRunner to wait for a specific area bitmap to appear.
GET TEXT FROM OBJECT/WINDOW	F11	Captures text in an object or a window.
GET TEXT FROM WINDOW AREA	Alt Right + F11	Captures text in a specified area and adds an **obj_get_text** statement to the test script.

Command	Default Softkey Combination	Function
GET TEXT FROM SCREEN AREA	Ctrl Right + F11	Captures text in a specified area and adds a **get_text** statement to the test script.
INSERT FUNCTION FOR OBJECT/WINDOW	F8	Inserts a TSL function for a GUI object.
INSERT FUNCTION FROM FUNCTION GENERATOR	F7	Opens the Function Generator dialog box.
STOP	Ctrl Left + F3	Stops test recording.
MOVE LOCATOR	Alt Left + F6	Records a **move_locator_abs** statement with the current position (in pixels) of the screen pointer.

Books Online

Find

Find Again

Help

Top of Chapter

Back

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 20, **Generating Functions**.

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.

Books Online

Find

Find Again

Help

Top of Chapter

Back
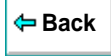
Editing a Test

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.
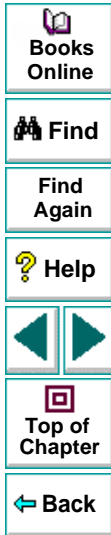
Managing Test Files

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

Creating a New Test

Choose **File** > **New** or click **New**. 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 **File** > **Open** or click **Open**.
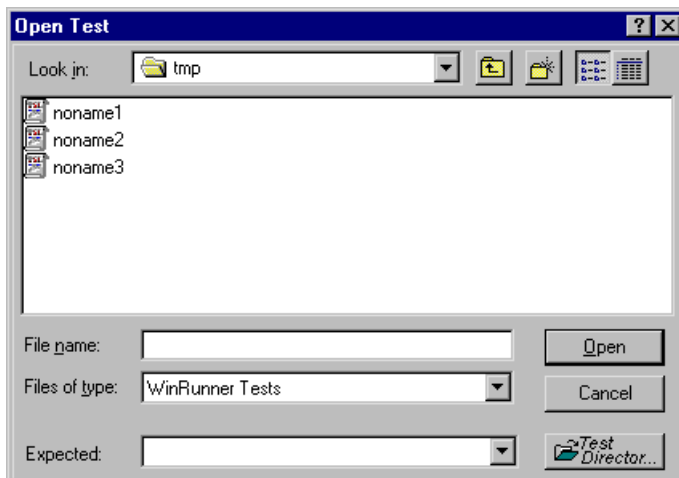

Note: No more than 100 tests may be open at the same time.

To open a test from the file system:

1 Choose **File** > **Open** or click **Open** 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 **File name** box, click the name of the test to open.

4 If the test has more than one set of expected results, click the folder you want to use on the **Expected** list. The default folder is called *exp.*
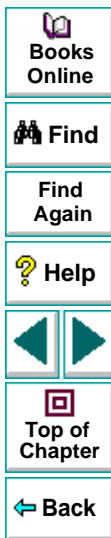
5 Click **Open** to open the test.
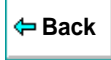
To open a test from a TestDirector project:

1 Choose **File** > **Open** or click **Open**. If you are connected to a TestDirector project, the **Open Test from TestDirector Project** dialog box opens and displays the test plan tree.
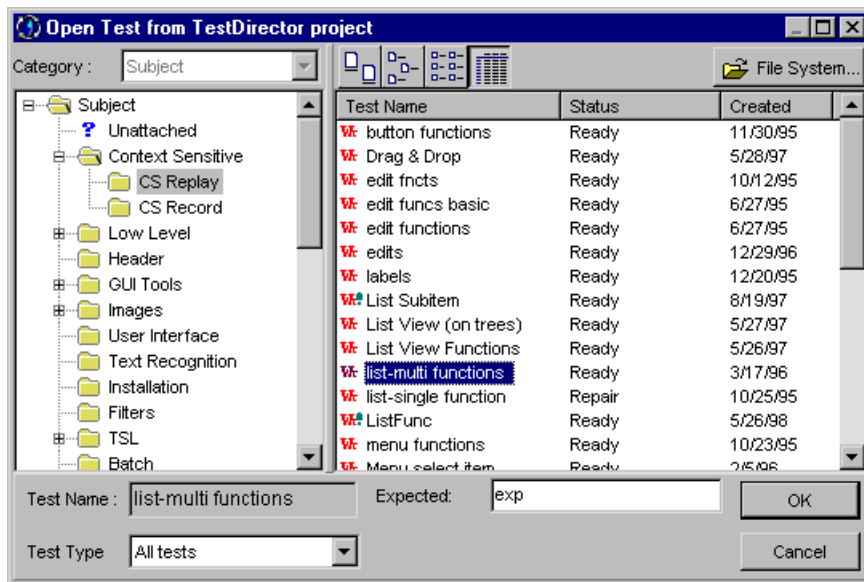
Note that the **Open Test from TestDirector Project** dialog box opens only when WinRunner is connected to a TestDirector project.

2 Click the relevant subject in the test plan tree. To expand the tree and view sublevels, double-click closed folders. To collapse the tree, double-click open folders.

Note that when you select a subject, the tests that belong to the subject appear in the Test Name list.

3 Select a test in the **Test Name** list. The test appears in the read-only **Test Name** box.

4 If desired, enter an expected results folder for the test in the **Expected** box. (Otherwise, the default folder 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 project, see Chapter 40, **Managing the Testing Process**.

Saving a Test

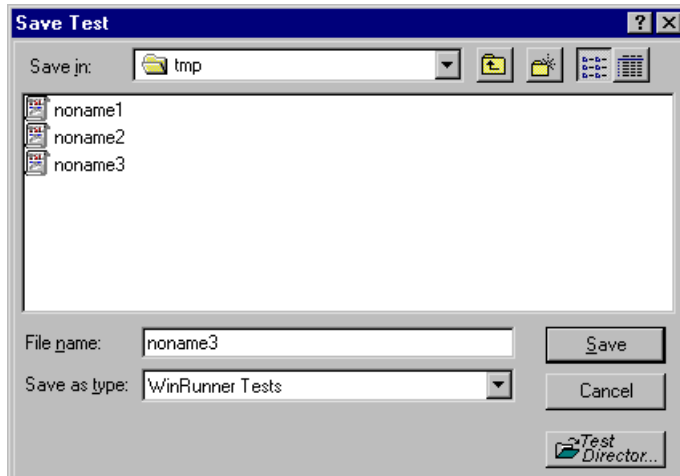The following options are available for saving tests:

- Save changes to a previously saved test by choosing **File** > **Save** or by clicking **Save**.

- Save two or more open tests simultaneously by choosing **File** > **Save All**.

- Save a new test script by choosing **File** > **Save As** or by clicking **Save**.

Books Online

Find

Find Again

Help

Top of Chapter

Back

To save a test to the file system:

🖫 **1** On the **File** menu, choose a **Save** command or click **Save**, as described above. The **Save Test** dialog box opens.
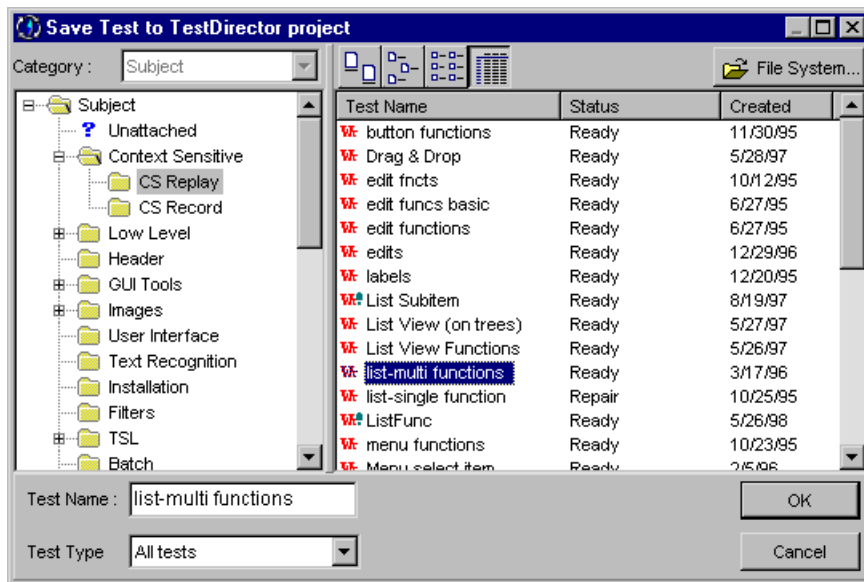
Save Test	? ×

Save in: 📁 tmp   ▼ 🔁 📑 ▦ ▦

- 📄 noname1
- 📄 noname2
- 📄 noname3

File name: noname3   [Save]

Save as type: WinRunner Tests ▼   [Cancel]

[📂 Test Director...]

2 In the **Save 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 project:

 1 On the **File** menu, choose a **Save** command or click **Save**, as described above. If you are connected to a TestDirector project, the **Save Test to TestDirector Project** dialog box opens.

Note that the **Save Test to TestDirector Project** dialog box opens only when WinRunner is connected to a TestDirector project.
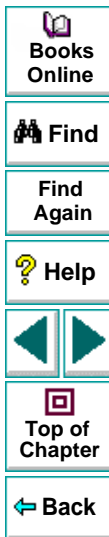
2 Select the relevant subject in the test plan tree. To expand the tree and view a sublevel, double-click a closed folder. To collapse a sublevel, double-click an open folder.

3 In the **Test Name** 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 project, see Chapter 40, **Managing the Testing Process**.

Printing a Test

To print a test script, choose **File** > **Print** 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 **File** > **Close**.

● To simultaneously close two or more open tests, choose **Window** > **Close All**.

By adding GUI checkpoints to your test scripts, you can compare the behavior of GUI objects in different versions of your application.
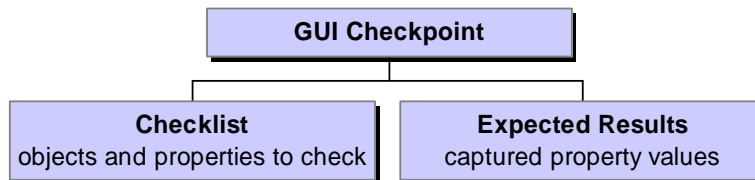
This chapter describes:

- **Checking a Single Property Value**
- **Checking a Single Object**
- **Checking Two or More Objects in a Window**
- **Checking All Objects in a Window**
- **Understanding GUI Checkpoint Statements**
- **Using an Existing GUI Checklist in a GUI Checkpoint**
- **Modifying GUI Checklists**
- **Understanding the GUI Checkpoint Dialog Boxes**
- **Property Checks and Default Checks**
- **Specifying Arguments for Property Checks**
- **Editing the Expected Value of a Property**
- **Modifying the Expected Results of a GUI Checkpoint**

Books Online

Find

Find Again

? Help

Top of Chapter

Back

About Checking GUI Objects

You can use GUI checkpoints in your test scripts to help you examine GUI objects in your application and detect defects. For example, you can check that when a specific dialog box opens, the OK, Cancel, and Help buttons are enabled.

You point to GUI objects and choose the properties 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.

Books Online

🔍 Find

Find Again

❓ Help

🔲 Top of Chapter

⬅ Back

```
                    ┌──────────────────────┐
                    │   GUI Checkpoint     │
                    └──────────────────────┘
                    ┌───────────┴───────────┐
        ┌─────────────────────┐   ┌─────────────────────┐
        │     Checklist       │   │   Expected Results  │
        │ objects and         │   │ captured property   │
        │ properties to check │   │ values              │
        └─────────────────────┘   └─────────────────────┘
```

When you run the test, the GUI checkpoint 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 28, **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 additional information.

You can use a regular expression to create a GUI checkpoint on an edit object or a static text object with a variable name. For additional information, see Chapter 26, **Using Regular Expressions**.

WinRunner provides special built-in support for ActiveX control, Visual Basic, and PowerBuilder application development environments. When you load the appropriate add-in support, WinRunner recognizes these controls, and treats them as it treats standard GUI objects. You can create GUI checkpoints for these objects as you would create them for standard GUI objects. WinRunner provides additional special built-in support for checking ActiveX and Visual Basic sub-objects. For additional information, see Chapter 10, **Working with ActiveX and Visual Basic Controls**. For information on WinRunner support for PowerBuilder, see Chapter 11, **Checking PowerBuilder Applications**.

You can also create GUI checkpoints that check the contents and properties of tables. For information, see Chapter 12, **Checking Table Contents**.

Note for XRunner users: 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 14, **Checking Bitmaps**.

Checking a Single Property Value

You can check a single 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, use the Check Property dialog box to add one of the following functions to the test script:

button_check_info   **scroll_check_info**

edit_check_info   **static_check_info**

list_check_info   **win_check_info**

obj_check_info

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

To create a GUI checkpoint for a property value:

1 Choose **Create > GUI Checkpoint > For Single Property**. If you are recording in Analog mode, press the CHECK GUI FOR SINGLE PROPERTY softkey in order to avoid extraneous mouse movements.

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

Books Online

M Find

Find Again

? Help

Top of Chapter

← Back

2 Click an object.

The Check Property dialog box opens and shows the default function for the selected object. WinRunner automatically assigns argument values to the function.

Check Property		⊠
button_check_info("OK","enabled",1);		Close
button	"OK"	☞ Paste
attribute	"enabled" ▼	Change >>
expected	1	
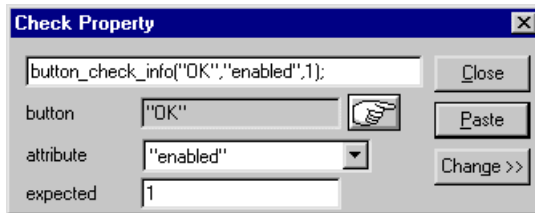
3 You can modify the arguments for the property check.

- To modify assigned argument values, choose a value from the **Attribute** list. The expected value is updated in the Expected text box.

- To choose a different object, click the pointing hand and then click an object in your application. WinRunner automatically assigns new argument values to the function.

    Note that if you click an object that is not compatible with the selected function, a message states that the current function cannot be applied to the selected object. Click OK to clear the message, and then click Close to close the Check Property dialog box. Repeat steps *1* and *2*.

4 Click **Paste** to paste the statement into your test script.

The function is pasted into the script at the insertion point. The Check Property dialog box closes.

Note: To change to another function for the object, click **Change**. The Function Generator dialog box opens and displays a list of functions. For more information on using the Function Generator, see Chapter 20, **Generating Functions**.

Checking a Single Object

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

Each standard object class has a set of default checks. For a complete list of standard objects, the properties you can check, and default checks, see **Property Checks and Default Checks** on page 241.

Note: You can set the default checks for an object using the **gui_ver_set_default_checks** function. For more information, refer to the *TSL Online Reference* and the *WinRunner Customization Guide*.

Creating a GUI Checkpoint using the Default Checks

You can create a GUI checkpoint that performs a default check on the property 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 using default checks:

1 Choose **Create** > **GUI Checkpoint > For Object/Window**, or click the **GUI Checkpoint for Object/Window** button on the User toolbar. If you are recording in Analog mode, press the CHECK GUI FOR OBJECT/WINDOW softkey in order to avoid extraneous mouse movements. Note that you can press the CHECK GUI FOR OBJECT/WINDOW softkey in Context Sensitive mode as well.

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

2 Click an object.

3 WinRunner captures the current value of the property of the GUI object being checked and stores it in the test's expected results folder. The WinRunner window is restored and a GUI checkpoint is inserted in the test script as an **obj_check_gui** statement. For more information, see **Understanding GUI Checkpoint Statements** on page 209.

Creating a GUI Checkpoint by Specifying which Properties to Check

You can specify which properties to check for an object. 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 **Create** > **GUI Checkpoint > For Object/Window**, or click the **GUI Checkpoint for Object/Window** button on the User toolbar. If you are recording in Analog mode, press the CHECK GUI FOR OBJECT/WINDOW softkey in order to avoid extraneous mouse movements. Note that you can press the CHECK GUI FOR OBJECT/WINDOW softkey in Context Sensitive mode as well.

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

Books Online

Find

Find Again

Help

Top of Chapter

Back
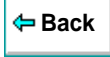
2 Double-click the object or window. The **Check GUI** dialog box opens.
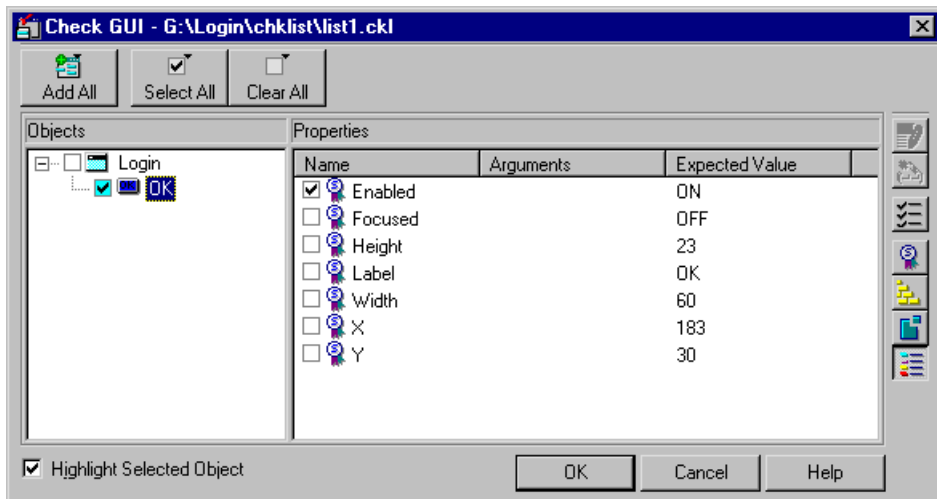
3 Click an object name in the **Objects** pane. The **Properties** pane lists all the properties for the selected object.

4 Select the properties you want to check.

- To edit the expected value of a property, first select it. Next, either click the **Edit Expected Value** button, or double-click the value in the **Expected Value** column to edit it. For more information, see **Editing the Expected Value of a Property** on page 263.

- To add a check in which you specify arguments, first select the property for which you want to specify arguments. Next, either click the **Specify Arguments** button, or double-click in the **Arguments** column. Note that if three dots appear in the Arguments column, then you must specify arguments for a check on this property. (You do not need to specify arguments if a default argument is specified.) When checking standard objects, you only specify arguments for certain properties of edit and static text objects. You also specify arguments for checks on certain properties of nonstandard objects. For more information, see **Specifying Arguments for Property Checks** on page 252.

- To change the viewing options for the properties of an object, use the Show Properties buttons. For more information, see **The Check GUI Dialog Box** on page 227.

5 Click **OK** to close the Check GUI dialog box.

WinRunner captures the GUI information and stores it in the test's expected results folder. 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 209.

For more information on the Check GUI dialog box, see **Understanding the GUI Checkpoint Dialog Boxes** on page 224.

Books Online

Find

Find Again

? Help

Top of Chapter

Back

Checking Two or More Objects in a Window

You can use a GUI checkpoint to check two or more objects in a window. For a complete list of standard objects and the properties you can check, see **Property Checks and Default Checks** on page 241.

To create a GUI checkpoint for two or more objects:

1 Choose **Create** > **GUI Checkpoint** > **For Multiple Objects** or click the **GUI Checkpoint for Multiple Objects** button on the User toolbar. If you are recording in Analog mode, press the CHECK GUI FOR MULTIPLE OBJECTS softkey in order to avoid extraneous mouse movements. 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. For more information on checking all objects in a window, see **Checking All Objects in a Window** on page 205.

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.
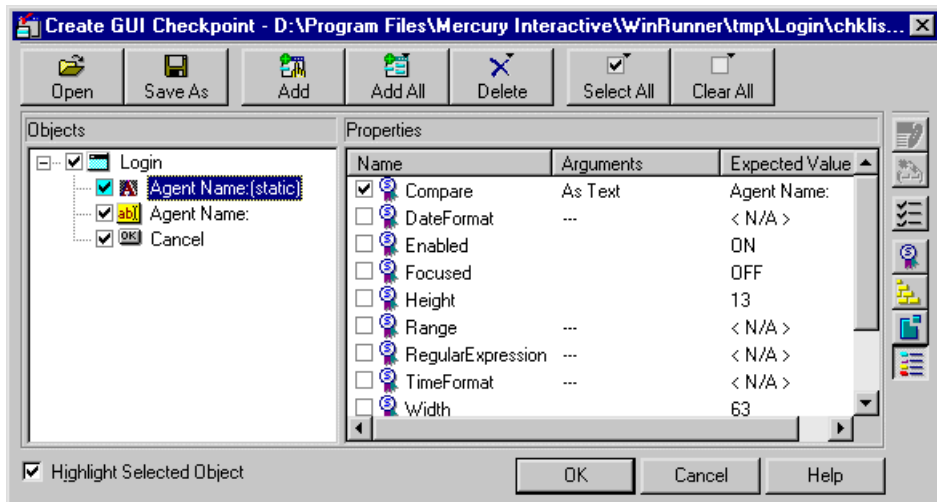
6 The Objects pane contains the name of the window and objects included in the GUI checkpoint. To specify which objects to check, click an object name in the **Objects** pane.

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

- To edit the expected value of a property, first select it. Next, either click the **Edit Expected Value** button, or double-click the value in the **Expected Value** column to edit it. For more information, see **Editing the Expected Value of a Property** on page 263.

- To add a check in which you specify arguments, first select the property for which you want to specify arguments. Next, either click the **Specify Arguments** button, or double-click in the **Arguments** column. Note that if three dots appear in the Arguments column, then you must specify arguments for a check on this property. (You do not need to specify arguments if a default argument is specified.) When checking standard objects, you only specify arguments for certain properties of edit and static text objects. You also specify arguments for checks on certain properties of nonstandard objects. For more information, see **Specifying Arguments for Property Checks** on page 252.

- To change the viewing options for the properties of an object, use the Show Properties buttons. For more information, see **The Create GUI Checkpoint Dialog Box** on page 231.

Books Online

Find

Find Again

Help

Top of Chapter

Back

7 To save the checklist and close the Create GUI Checkpoint dialog box, click **OK**.

WinRunner captures the current property values of the selected GUI objects and stores it in the expected results folder. A **win_check_gui** statement is inserted in the test script. For more information, see **Understanding GUI Checkpoint Statements** on page 209.

For more information on the Create GUI Checkpoint dialog box, see **Understanding the GUI Checkpoint Dialog Boxes** on page 224.
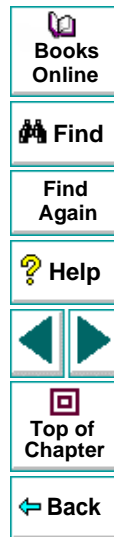
Checking All Objects in a Window

You can create a GUI checkpoint to perform default checks on all GUI objects in a window. Alternatively, you can specify which checks to perform on all GUI objects in a window.

Each standard object class has a set of default checks. For a complete list of standard objects, the properties you can check, and default checks, see **Property Checks and Default Checks** on page 241.

Note: You can set the default checks for an object using the **gui_ver_set_default_checks** function. For more information, refer to the *TSL Online Reference* and the *WinRunner Customization Guide*.

Books Online

M Find

Find Again

? Help

Top of Chapter

Back

Checking All Objects in a Window using Default Checks

You can create a GUI checkpoint that checks the default property of every GUI object in a window.

To create a GUI checkpoint that performs a default check on every GUI object in a window:

1 Choose **Create** > **GUI Checkpoint** > **For Object/Window**, or click the **GUI Checkpoint for Object/Window** button on the User toolbar. If you are recording in Analog mode, press the CHECK GUI FOR OBJECT/WINDOW softkey in order to avoid extraneous mouse movements. Note that you can press the CHECK GUI FOR OBJECT/WINDOW softkey in Context Sensitive mode as well.

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

2 Click the title bar or the menu bar of the window you want to check.

The **Add All** dialog box opens.
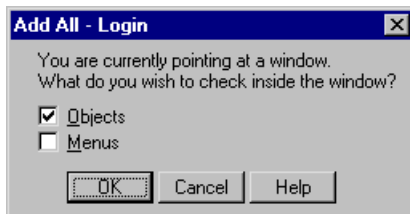
3 Select **Objects**, **Menus**, or both to indicate the types of objects to include in the checklist. When you select only Objects (the default setting), all objects in the window except for menus are included in the checklist. To include menus in the checklist, select Menus.

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

WinRunner captures the expected property values of the GUI objects and/or menu items and stores this information in the test's expected results folder. The WinRunner window is restored and a **win_check_gui** statement is inserted in the test script.
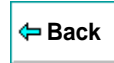
Specifying which Checks to Perform on All Objects in a Window

You can use a GUI checkpoint to specify which checks to perform on all GUI objects in a window.

To create a GUI checkpoint in which you specify which checks to perform on all GUI objects in a window:

 1 Choose **Create** > **GUI Checkpoint** > **For Object/Window**, or click the **GUI Checkpoint for Object/Window** button on the User toolbar. If you are recording in Analog mode, press the CHECK GUI FOR OBJECT/WINDOW softkey in order to avoid extraneous mouse movements. Note that you can press the CHECK GUI FOR OBJECT/WINDOW softkey in Context Sensitive mode as well.

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

 2 Double-click the title bar or the menu bar of the window you want to check.

WinRunner generates a new checklist containing all the objects in the window. This may take a few seconds.

The Check GUI dialog box opens. Specify which checks to perform, and click **OK** to close the dialog box. For more information, see **The Check GUI Dialog Box** on page 227.

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

Understanding GUI Checkpoint Statements

A GUI checkpoint for a single object appears in your script as an **obj_check_gui** statement. A GUI checkpoint that checks more than one object in a window appears in your script as a **win_check_gui** statement. Both the **obj_check_gui** and **win_check_gui** statements are always associated with a *checklist* and store expected results in a *expected results file*.

- A *checklist* lists the objects and properties that need to be checked. For an **obj_check_gui** statement, the checklist lists only one object. For a **win_check_gui** statement, a checklist contains a list of all objects to be checked in a window. When you create a GUI checkpoint, you can create a new checklist or use an existing checklist. For information on using an existing checklist, see **Using an Existing GUI Checklist in a GUI Checkpoint** on page 212.

- An *expected results file* contains the expected property values for each object in the checklist. These property values are captured when you create a checkpoint, and can later be updated manually or by running the test in Update mode. For more information, see **Running a Test to Update Expected Results** on page 689. Each time you run the test, the expected property values are compared to the current property values of the objects.

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

obj_check_gui (*object*, *checklist*, *expected results file*, *time*);

Books Online

M Find

Find Again

? Help

Top of Chapter

Back

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 *expected results file* is the name of the file that stores the expected property values. The *time* is the interval marking the maximum delay between the previous input event and the capture of the current property values, in seconds. This interval is added to the *timeout_msec* testing option during the test run. For more information on the *timeout_msec* testing option, see Chapter 37, **Setting Testing Options from a Test Script**.

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);

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

win_check_gui (*window*, *checklist*, *expected results file*, *time*);

The *window* is the logical name of the GUI window. The *checklist* is the name of the checklist defining the objects and properties to check. The *expected results file* is the name of the file that stores the expected property values. The *time* is the interval marking the maximum delay between the previous input event and the capture of the current property values, in seconds. This interval is added to the *timeout_msec* testing option during the test run. For more information on the *timeout_msec* testing option, see Chapter 37, **Setting Testing Options from a Test Script**.

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*.

Using an Existing GUI Checklist in a GUI Checkpoint

You can create a GUI checkpoint using an existing GUI checklist. This is useful when you want to use a GUI checklist to create new GUI checkpoints, either in your current test or in a different test. For example, you may want to check the same properties of certain objects at several different points during your test. These object properties may have different expected values, depending on when you check them.

Although you can create a new GUI checklist whenever you create a new GUI checkpoint, it is expedient to "reuse" a GUI checklist in as many checkpoints as possible. Using a single GUI checklist in many GUI checkpoints facilitates the testing process by reducing the time and effort involved in maintaining the GUI checkpoints in your test.

In order for WinRunner to locate the objects to check in your application, you must load the appropriate GUI map file before you run the test. For information about loading GUI map files, see **Loading the GUI Map File** on page 78.

Books Online

Find

Find Again

Help

Top of Chapter

Back

Note: If you want a checklist to be available to more than one test, you must save it in a shared folder. For information on saving a GUI checklist in a shared folder, see **Saving a GUI Checklist in a Shared Folder** on page 216.

To use an existing GUI checklist in a GUI checkpoint:

1 Choose **Create** > **GUI Checkpoint** > **For Multiple Objects** or click the **GUI Checkpoint for Multiple Objects** button on the User toolbar.

The Create GUI Checkpoint dialog box opens.

2 Click **Open**.

The Open Checklist dialog box opens.

3 To see checklists in the Shared folder, click **Shared**.

Open Checklist

Enter Checklist Name:

list1.ckl

list1.ckl
list2.ckl
list3.ckl

OK

Cancel

Help

Scope

○ Test
○ Shared

Enter Checklist Description:

Checklist for window "Login"
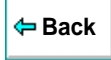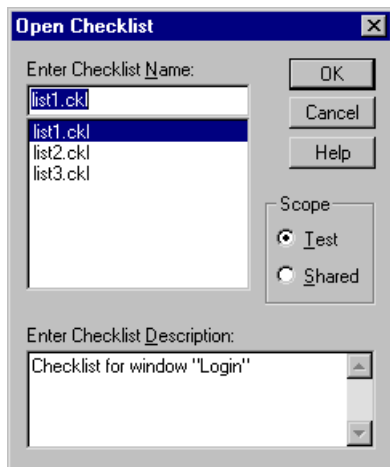
4 Select a checklist and click **OK**.

The Open Checklist dialog box closes and the selected list appears in the Create GUI Checkpoint dialog box.

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

6 Click **OK**. WinRunner captures the current property values and a **win_check_gui** statement is inserted into your test script.

Modifying GUI Checklists

You can make changes to a checklist you created for a GUI checkpoint. Note that a checklist includes only the objects and properties that need to be checked. It does not include the expected results for the values of those properties.

You can:

- make a checklist available to other users by saving it in a shared folder
- edit a checklist

Note: In addition to modifying GUI checklists, you can also modify the expected results of GUI checkpoints. For more information, see **Modifying the Expected Results of a GUI Checkpoint** on page 266.

Saving a GUI Checklist in a Shared Folder

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

The default folder in which WinRunner stores your shared checklists is *WinRunner installation folder/chklist*. To choose a different folder, you can use the **Shared Checklists** box in the Folders tab of the General Options dialog box. For more information, see Chapter 36, **Setting Global Testing Options**.

To save a GUI checklist in a shared folder:

 1 Choose **Create** > **Edit GUI Checklist**.

The Open Checklist dialog box opens. Note that GUI checklists have the .ckl extension, while database checklists have the .cdl extension. For information on database checklists, see **Modifying a Database Checkpoint** on page 371.

 2 Select a GUI checklist and click **OK**.

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

Books Online

Find

Find Again

Help

Top of Chapter

Back

3 Save the checklist by clicking **Save As**.

The Save Checklist dialog box opens.
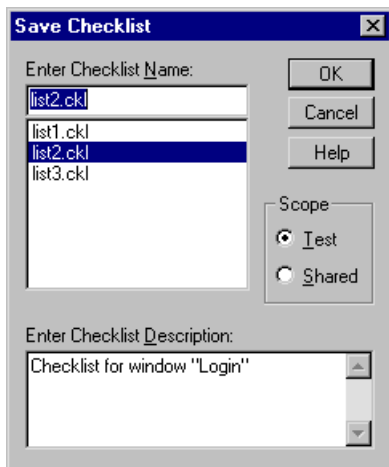
4 Under **Scope**, click **Shared**. Type in a name for the shared checklist. Click **OK** to save the checklist and close the dialog box.

5 Click **OK** to close the Edit GUI Checklist dialog box.
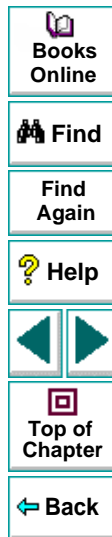
Editing GUI Checklists

You can edit an existing GUI checklist. Note that a GUI checklist includes only the objects and the properties to be checked. It does not include the expected results for the values of those properties.

You may want to edit a GUI checklist if you add a checkpoint for a window that already has a checklist.

When you edit a GUI checklist, you can:

● change which objects in a window to check

● change which properties of an object to check

● change the arguments for an existing property check

● specify the arguments for a new property check

Note that before you start working, the objects in the checklist must be loaded into the GUI map. For information about loading the GUI map, see **Loading the GUI Map File** on page 78.
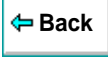
To edit an existing GUI checklist:

1 Choose **Create** > **Edit GUI Checklist**. The Open Checklist dialog box opens.

2 A list of checklists for the current test is displayed. If you want to see checklists in a shared folder, click **Shared**.

For more information on sharing GUI checklists, see **Saving a GUI Checklist in a Shared Folder** on page 216.

Lists the available checklists.

Displays checklists created for the current test.

Displays checklists created in a shared folder.

Describes the selected checklist.
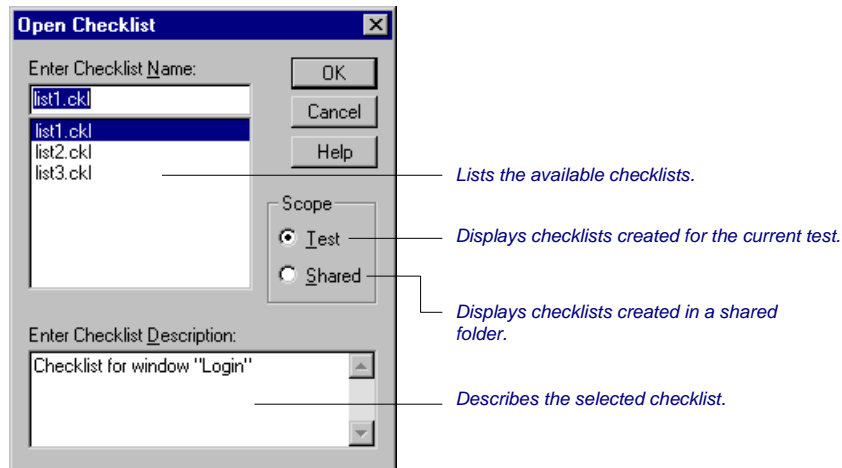
3 Select a GUI checklist.

4 Click **OK**.

The Open Checklist dialog box closes. The Edit GUI Checklist dialog box opens and displays the selected checklist.
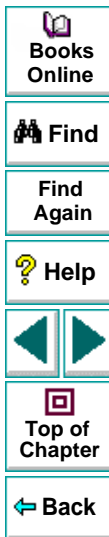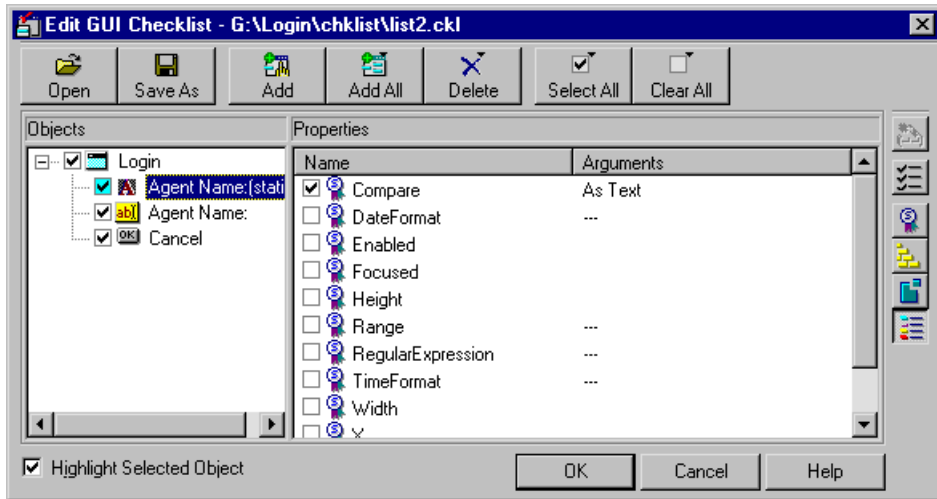
5 To see a list of the properties to check for a specific object, click the object name in the **Objects** pane. The **Properties** pane lists all the properties for the selected object. To change the viewing options for the properties for an object, use the Show Properties buttons. For more information, see **The Edit GUI Checklist Dialog Box** on page 236.

- To check additional properties of an object, select the object in the **Objects** pane. In the **Properties** pane, select the properties to be checked.

- To delete an object from the checklist, select the object in the **Objects** pane. 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.

  In the **Properties** pane, select the properties you want to check or accept the default checks.

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

Add All
- To add all objects or menus in a window 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**.

Note: If the edited checklist is part of an **obj_check_gui** statement, do not add additional objects to it, as by definition this statement is for a single object only.

- To add a check in which you specify arguments, first select the property for which you want to specify arguments. Next, either click the **Specify Arguments** button, or double-click in the **Arguments** column. Note that if three dots appear in the Arguments column, then you must specify arguments for a check on this property. (You do not need to specify arguments if a default argument is specified.) When checking standard objects, you only specify arguments for certain properties of edit and static text objects. You also specify arguments for checks on certain properties of nonstandard objects. For more information, see **Specifying Arguments for Property Checks** on page 252.

Books Online

Find

Find Again

Help

Top of Chapter

Back

6 Save the checklist in one of the following ways:

- To save the checklist under its existing name, click **OK** to close the Edit GUI Checklist dialog box. A WinRunner message prompts you to overwrite the existing checklist. Click **OK**.

- To save the checklist under a different name, 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 default name when you click OK to close the Edit GUI Checklist dialog box.

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 224.

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 Run Modes** on page 677.

Understanding the GUI Checkpoint Dialog Boxes

When creating a GUI checkpoint to check your GUI objects, you can specify the objects and properties to check, create new checklists, and modify existing checklists. Three dialog boxes are used to create and maintain your GUI checkpoints: the *Check GUI* dialog box, the *Create GUI Checkpoint* dialog box, and the *Edit GUI Checklist* dialog box.

Note that by default, the toolbar at the top of each GUI Checkpoint dialog box displays large buttons with text. You can choose to see dialog boxes with smaller buttons without titles. Examples of both kinds of buttons are illustrated below.

Large Add All button   *Small Add All button*

To display the GUI Checkpoint dialog boxes with small buttons:

1 Click the top-left corner of the dialog box.

2 Clear the **Large Buttons** option.

Messages in the GUI Checkpoint Dialog Boxes

The following messages may appear in the GUI Checkpoint dialog boxes:

Message	Meaning	Dialog Box	Location
Complex Value	The expected or actual value of the selected property check is too complex to display in the column. This message often appears for content checks on tables.	Check GUI , Create GUI Checkpoint, GUI Checkpoint Results* (see note below)	Properties pane, Expected Value column or Actual Value column
N/A	The expected value of the selected property check was not captured: either arguments need to be specified before this check can have an expected value, or the expected value of this check is captured only once this check is added to the checkpoint.	Check GUI , Create GUI Checkpoint, GUI Checkpoint Results* (see note below)	Properties pane, Expected Value column
Cannot Capture	The expected or actual value of the selected property could not be captured.	Check GUI , Create GUI Checkpoint, GUI Checkpoint Results* (see note below)	Properties pane, Expected Value column or Actual Value

Books Online

🔍 **Find**

Find Again

❔ **Help**

◀ ▶

▣ **Top of Chapter**

⬅ **Back**

Message	Meaning	Dialog Box	Location
No properties are available for this object	The specified object did not have any properties.	Check GUI , Create GUI Checkpoint, Edit GUI Checklist	Properties pane
No properties were captured for this object	When this checkpoint was created, no property checks were selected for this object.	GUI Checkpoint Results* (see note below)	Properties pane

Books Online

M Find

Find Again

? Help

Top of Chapter

Back

Note: For information on the GUI Checkpoint Results dialog box, see **Modifying the Expected Results of a GUI Checkpoint** on page 266 or Chapter 28, **Analyzing Test Results**.

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. This dialog box opens when you choose **Create** > **GUI Checkpoint** > **For Object/Window** or click the **GUI Checkpoint for Object/Window** button on the User toolbar, and double-click an object or a window.

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

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

**Note:** When arguments have not been specified for a property check that requires arguments, <N/A> appears in the **Expected Value** column for that check. The arguments specified for a check determine its expected value, and therefore the expected value is not available until the arguments are specified.

The Check GUI dialog box includes the following options:

Button	Description
Add All	**Add All** adds all objects or menus in a window to your checklist.
Select All	**Select All** selects all objects, properties, or objects of a given class in the Check GUI dialog box. If you want to select all objects of a given class, the Classes of Objects dialog box opens. Specify the class of objects to select.
Clear All	**Clear All** clears all objects, properties, or objects of a given class in the Check GUI dialog box. If you want to clear all objects of a given class, the Classes of Objects dialog box opens. Specify the class of objects to clear.
Property List	**Property List** calls the *ui_function* parameter that is defined only for classes customized using the **gui_ver_add_class** function. Note that this button appears only if at least one object in the Objects pane belongs to a class for which the *ui_function* parameter has been defined using the **gui_ver_add_class** function. For additional information, refer to the *WinRunner Customization Guide*.
	Edit Expected Value enables you to edit the expected value of the selected property. For more information, see **Editing the Expected Value of a Property** on page 263.

Button	Description
🔧	**Specify Arguments** enables you to specify the arguments for a check on the selected property. For more information, see **Specifying Arguments for Property Checks** on page 252.
🗒	**Show Selected Properties Only** displays only properties whose check boxes are selected. (Toggles between viewing all properties and viewing selected properties only.) By default, all properties are shown.
🏅	**Show Standard Properties Only** displays only standard properties.
📊	**Show Nonstandard Properties Only** displays only nonstandard properties, such as Visual Basic, PowerBuilder, and ActiveX control properties.
🔲	**Show User Properties Only** displays only user-defined property checks. To create user-defined property checks, refer to the *WinRunner Customization Guide.*
📋	**Show All Properties** displays all properties, including standard, nonstandard, and user-defined properties.

When you click OK to close the dialog box, WinRunner captures the current property values and stores them in the test's expected results folder. 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 **Create** > **GUI Checkpoint** > **For Multiple Objects** or click the **GUI Checkpoint for Multiple Objects** button on the User toolbar.

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

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

Note: When arguments have not been specified for a property check that requires arguments, <N/A> appears in the **Expected Value** column for that check. The arguments specified for a check determine its expected value, and therefore the expected value is not available until the arguments are specified.

Books Online

Find

Find Again

Help

Top of Chapter

Back

The Create GUI Checkpoint dialog box includes the following options:

| Button	Description
Open	**Open** opens an existing GUI checklist.
Save As	**Save As** saves the open GUI checklist to a different name. Note that if you do not click the Save As button, WinRunner automatically saves the checklist under its default name when you click OK to close the Create GUI Checkpoint dialog box. The Save As option is particularly useful for saving a checklist to the "shared checklist" folder.
Add	**Add** adds an object to your GUI checklist.
Add All	**Add All** adds all objects or menus in a window to your GUI checklist.
Delete	**Delete** deletes an object, or all of the objects that appear in the GUI checklist.
Select All	**Select All** selects all objects, properties, or objects of a given class in the Create GUI Checkpoint dialog box. If you want to select all objects of a given class, the Classes of Objects dialog box opens. Specify the class of objects to select.

Button	Description
Clear All	**Clear All** clears all objects, properties, or objects of a given class in the Create GUI Checkpoint dialog box. If you want to clear all objects of a given class, the Classes of Objects dialog box opens. Specify the class of objects to clear.
Property List	**Property List** calls the *ui_function* parameter that is defined only for classes customized using the **gui_ver_add_class** function. Note that this button appears only if at least one object in the Objects pane belongs to a class for which the *ui_function* parameter has been defined using the **gui_ver_add_class** function. For additional information, refer to the *WinRunner Customization Guide*.
	Edit Expected Value enables you to edit the expected value of the selected property. For more information, see **Editing the Expected Value of a Property** on page 263.
	Specify Arguments enables you to specify the arguments for a check on the selected property. For more information, see **Specifying Arguments for Property Checks** on page 252.
	Show Selected Properties Only displays only properties whose check boxes are selected. (Toggles between viewing all properties and viewing selected properties only.) By default, all properties are shown.
	Show Standard Properties Only displays only standard properties.

| Button	Description
	Show Nonstandard Properties Only displays only nonstandard properties, such as Visual Basic, PowerBuilder, and ActiveX control properties.
	Show User Properties Only displays only user-defined property checks. To create user-defined property checks, refer to the *WinRunner Customization Guide*.
	Show All Properties displays all properties, including standard, nonstandard, and user-defined properties.

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

Books Online

Find

Find Again

Help

Top of Chapter

Back
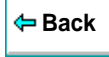
The Edit GUI Checklist Dialog Box

You can use the Edit GUI Checklist dialog box to modify your checklist. A checklist contains a list of objects and properties. It does not capture the current values for those properties. Consequently you cannot edit the expected values of an object's properties in this dialog box.

To open the Edit GUI Checklist dialog box, choose **Create > Edit GUI Checklist**.
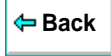
Books Online

M Find

Find Again

? Help

◀ ▶
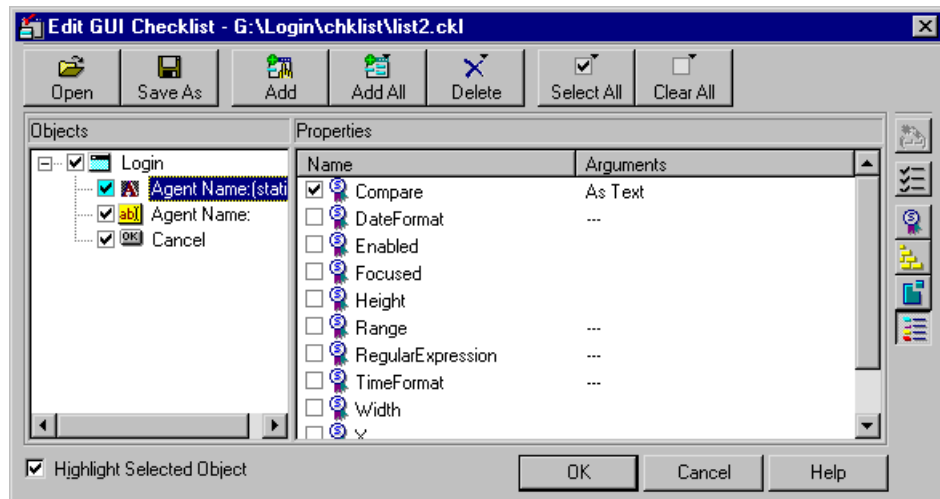
▣ **Top of Chapter**

⇐ **Back**

The **Objects** pane contains the name of the window and objects that are included in the checklist. The **Properties** pane 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.

When you select an object from the Objects pane, 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:

Button	Description
Open	**Open** opens an existing GUI checklist.
Save As	**Save As** saves your GUI checklist to another location. Note that if you do not click the Save As button, WinRunner will automatically save the checklist under its default name when you click OK to close the Edit GUI Checklist dialog box. This option is particularly useful for saving a checklist to the "shared checklist" folder.
Add	**Add** adds an object to your GUI checklist.
Add All	**Add All** adds all objects or all menus in a window to your GUI checklist.
Delete	**Delete** deletes the specified object, or all objects that appear in the GUI checklist.
Select All	**Select All** selects all objects, properties, or objects of a given class in the Edit GUI Checklist dialog box. If you want to select all objects of a given class, the Classes of Objects dialog box opens. Specify the class of objects to select.

Books Online

Find

Find Again

Help

Top of Chapter

Back

Button	Description
Clear All	**Clear All** clears all objects, properties, or objects of a given class in the Edit GUI Checklist dialog box. If you want to clear all objects of a given class, the Classes of Objects dialog box opens. Specify the class of objects to clear.
Property List	**Property List** calls the *ui_function* parameter that is defined only for classes customized using the **gui_ver_add_class** function. Note that this button appears only if at least one object in the Objects pane belongs to a class for which the *ui_function* parameter has been defined using the **gui_ver_add_class** function. For additional information, refer to the *WinRunner Customization Guide*.
	Specify Arguments enables you to specify the arguments for a check on the selected property. For more information, see **Specifying Arguments for Property Checks** on page 252.
	Show Selected Properties Only displays only properties whose check boxes are selected. (Toggles between viewing all properties and viewing selected properties only.) By default, selected properties are shown.
	Show Standard Properties Only displays only standard properties.
	Show Nonstandard Properties Only displays only nonstandard properties, such as Visual Basic, PowerBuilder, and ActiveX control properties.

| Button	Description
	Show User Properties Only displays only user-defined property checks. To create user-defined property checks, refer to the *WinRunner Customization Guide*.
	Show All Properties displays all properties, including standard, nonstandard, and user-defined properties.

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.

A new GUI checkpoint statement is *not* inserted in your test script.

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 Run Modes** on page 677.
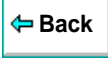
Property Checks and Default Checks

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, you can specify in a dialog box 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 **Create > GUI Checkpoint** command. 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 properties to check for an object, you choose a **Create > GUI Checkpoint** command. 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 types of checks available for different object classes.

Books Online

Find

Find Again

Help

Top of Chapter

Back

Calendar Class

You can check the following properties for a calendar class object:

Enabled: Checks whether the calendar can be selected.

Focused: Checks whether keyboard input will be directed to the calendar.

Height: Checks the calendar's height in pixels.

Selection: The selected date in the calendar (default check).

Width: Checks the calendar's width in pixels.

X: Checks the x-coordinate of the top left corner of the calendar, relative to the window.

Y: Checks the y-coordinate of the top left corner of the calendar, relative to the window.

Books Online

Find

Find Again

? Help

Top of Chapter

Back

Check_button Class and Radio_button Class

You can check the following properties for a check box (an object of check_button class) or a radio button:

Enabled: Checks whether the button can be selected.

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

Height: Checks the button's height in pixels.

Label: Checks the button's label.

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

Width: Checks the button's width in pixels.

X: Checks the x-coordinate of the top left corner of the button, relative to the window.

Y: Checks the y-coordinate of the top left corner of the button, relative to the window.

Edit Class and Static Text Class

You can check the properties below for edit class and static_text class objects.

Checks on any of these five properties (Compare, DateFormat, Range, RegularExpression, and TimeFormat) require you to specify arguments. For information on specifying arguments for property checks, see **Specifying Arguments for Property Checks** on page 252.

Compare: Checks the contents of the object (default check). This check has arguments. You can specify the following arguments:

- a case-sensitive check on the contents as text (default setting)
- a case-insensitive check on the contents as text
- or numeric check on the contents

DateFormat: Checks that the contents of the object are in the specified date format. You must specify arguments (a date format) for this check. WinRunner supports a wide range of date formats. For a complete list of available date formats, see **Date Formats** on page 255.

Enabled: Checks whether the object can be selected.

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

Height: Checks the object's height in pixels.
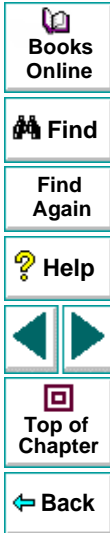
Range: Checks that the contents of the object are within the specified range. You must specify arguments (the upper and lower limits for the range) for this check.

RegularExpression: Checks that the string in the object meets the requirements of the regular expression. You must specify arguments (the string) for this check. Note that you do not need to precede the regular expression with an exclamation point. For more information, see Chapter 26, **Using Regular Expressions**.

TimeFormat: Checks that the contents of the object are in the specified time format. You must specify arguments (a time format) for this check. WinRunner supports the time formats shown below, with an example for each format.

hh.mm.ss	10.20.56
hh:mm:ss	10:20:56
hh:mm:ss ZZ	10:20:56 AM

Width: Checks the text object's width in pixels.

X: Checks the x-coordinate of the top left corner of the object, relative to the window.

Y: Checks the y-coordinate of the top left corner of the object, relative to the window.

List Class

You can check the following properties for a list object:

Content: Checks the contents of the entire list.

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

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

Height: Checks the list's height in pixels.

ItemsCount: Checks the number of items in the list.

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

Width: Checks the list's width in pixels.

X: Check the x-coordinate of the top left corner of the list, relative to the window.

Y: Check the y-coordinate of the top left corner of the list, relative to the window.

Menu_item Class

Menus cannot be accessed directly, by clicking them. To include a menu in a GUI checkpoint, click the window title bar or the menu bar. The **Add All** dialog box opens. Select the **Menus** option. All menus in the window are added to the checklist. Each menu item is listed separately.

You can check the following properties for menu items:

HasSubMenu: Checks whether a menu item has a submenu.

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

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

SubMenusCount: Counts the number of items in the submenu.

Object Class

You can check the following properties for an object that is not mapped to a standard object class:

Enabled: Checks whether the object can be selected.
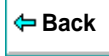
Focused: Checks whether keyboard input will be directed to this object.

Height: Checks the object's height in pixels (default check).

Width: Checks the object's width in pixels (default check).

X: Checks the x-coordinate of the top left corner of the GUI object, relative to the window (default check).

Y: Checks the y-coordinate of the top left corner of the GUI object, relative to the window (default check).

Push_button Class

You can check the following properties for a push button:

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

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

Height: Checks the button's height in pixels.

Label: Checks the button's label.

Width: Checks the button's width in pixels.

X: Checks the x-coordinate of the top left corner of the button, relative to the window.

Y: Checks the y-coordinate of the top left corner of the button, relative to the window.

Scroll Class

You can check the following properties for a scrollbar:

Enabled: Checks whether the scrollbar can be selected.

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

Height: Checks the scrollbar's height in pixels.

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

Width: Checks the scrollbar's width in pixels.

X: Checks the x-coordinate of the top left corner of the scrollbar, relative to the window.

Y: Checks the y-coordinate of the top left corner of the scrollbar, relative to the window.

Books Online

Find

Find Again

Help

Top of Chapter

Back

Window Class

You can check the following properties for a window:

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

Enabled: Checks whether the window can be selected.

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

Height: Checks the window's height in pixels.

Label: Checks the window's label.

Maximizable: Checks whether the window can be maximized.

Maximized: Checks whether the window is maximized.

Minimizable: Checks whether the window can be minimized.

Minimized: Checks whether the window is minimized.

Resizable: Checks whether the window can be resized.

SystemMenu: Checks whether the window has a system menu.

Width: Checks the window's width in pixels.

X: Checks the x-coordinate of the top left corner of the window.

Y: Checks the y-coordinate of the top left corner of the window.

Specifying Arguments for Property Checks

You can perform many different property checks on objects. If you want to perform the property checks listed below on edit class and static_text class objects, you must specify arguments for those checks:

- Compare

- DateFormat

- Range

- RegularExpression

- TimeFormat

To specify arguments for a property check on an edit class or static_text class object:

1 Make sure that one of the GUI Checkpoint dialog boxes containing the object for whose property you want to specify arguments is open. If necessary, choose **Create > GUI Checkpoint > For Multiple Objects** or **Create > Edit GUI Checklist** to open the relevant dialog box.

2 In the **Objects** pane of the dialog box, select the object to check.

3 In the **Properties** pane of the dialog box, select the desired property check.

4 Do one of the following:

● Click the **Specify Arguments** button.

● Double-click the default argument (for the Compare check) or the three dots in the corresponding **Arguments** column (for the other checks).

● Right-click with the mouse and choose **Specify Arguments** from the pop-up menu.

A dialog box for the selected property check opens.

Note: When you select the check box beside a property check for which you need to specify arguments, the dialog box for the selected property check opens automatically.

5 Specify the arguments in the dialog box that opens. For example, for a Date Format check, specify the date format. For information on specifying arguments for a particular property check, see the relevant section below.

6 Click OK to close the dialog box for specifying arguments.

7 When you are done, click OK to close the GUI Checkpoint dialog box that is open.

Books Online

Find

Find Again

Help

Top of Chapter

Back
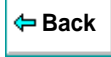
Compare Property Check

Checks the contents of the edit class or static_text class object (default check). Opens the **Specify 'Compare' Arguments** dialog box.
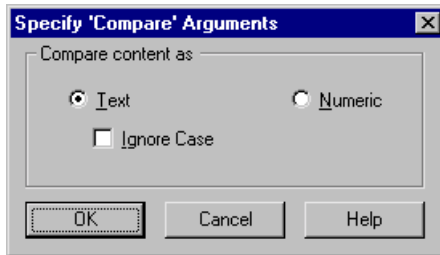
- Click **Text** to check the contents as text (default setting).

- To ignore the case when checking text, select the **Ignore Case** check box.

- Click **Numeric** to check the contents as a number.

Note that the default argument setting for the Compare property check is a case-sensitive comparison of the object as text.

DateFormat Property Check

Checks that the contents of the edit or static_text class object are in the specified date format. To specify a date format, select it from the drop-down list in the Check Arguments dialog box.
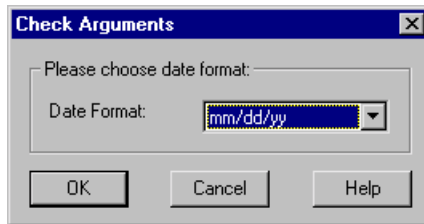
Date Formats

WinRunner supports the following date formats, shown with an example for each:

mm/dd/yy	09/24/99
dd/mm/yy	24/09/99
dd/mm/yyyy	24/09/1999
yy/dd/mm	99/24/09
dd.mm.yy	24.09.99
dd.mm.yyyy	24.09.1999
dd-mm-yy	24-09-99

dd-mm-yyyy	24-09-1999
yyyy-mm-dd	1999-09-24
Day, Month dd, yyyy	Friday (or Fri), September (or Sept) 24, 1999
dd Month yyyy	24 September 1999
Day dd Month yyyy	Friday (or Fri) 24 September (or Sept) 1999

Note: When the day or month begins with a zero (such as 09 for September), the 0 is not required for a successful format check.
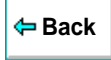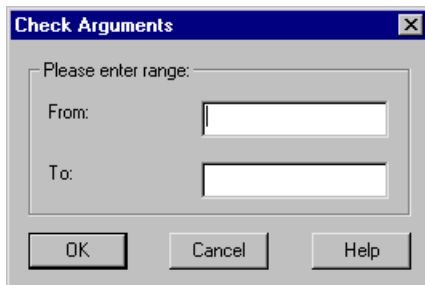
Range Property Check

Checks that the contents of the edit class or static_text class object are within the specified range. In the Check Arguments dialog box, specify the lower limit in the top edit field, and the upper limit in the bottom edit field.

Note: Any currency sign preceding the number is removed prior to making the comparison for this check.

Books Online

Find

Find Again

Help

Top of Chapter

Back

RegularExpression Property Check

Checks that the string in the edit class or static_text class object meets the requirements of the regular expression. In the Check Arguments dialog box, enter a string into the Regular Expression box. You do not need to precede the regular expression with an exclamation point. For more information, see Chapter 26, **Using Regular Expressions**.
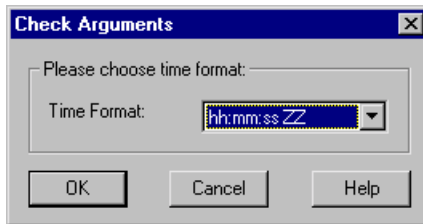
Check Arguments

Please enter regular expression:

Regular
Expression:

| OK | Cancel | Help |
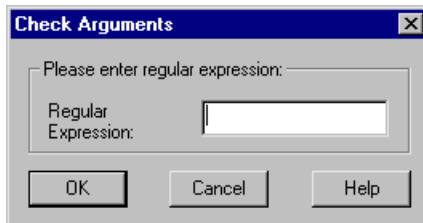
Note: Two "\" characters ("\\") are interpreted as a single "\" character.

TimeFormat Property Check

Checks that the contents of the edit class or static_text class object are in the specified time format. To specify the time format, select it from the drop-down list in the Check Arguments dialog box.

Books Online

Find

Find Again

? Help

WinRunner supports the following time formats, shown with an example for each:

Top of Chapter

← Back

Time Formats

hh.mm.ss	10.20.56
hh:mm:ss	10:20:56
hh:mm:ss ZZ	10:20:56 AM
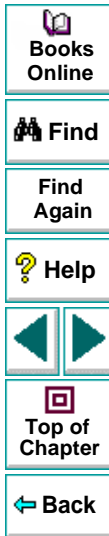
Closing the GUI Checkpoint Dialog Boxes

If you select property checks that requires arguments without specifying the actual arguments for them, and then click OK to close the dialog box, you are prompted to specify the arguments.

Specifying Arguments for One Property Check

If you click OK to close a GUI checkpoint dialog box when you have selected a check on a property that requires arguments, without first specifying arguments for that property check, the Check Arguments dialog box for that property check opens.

Specifying Arguments for Multiple Property Checks

If you select check boxes for multiple property checks that need arguments, and you did not specify arguments, then when you try to close to open dialog box, the Argument Specification dialog box opens. This dialog box enables you to specify arguments for the relevant property checks.

In the example below, the user clicked OK to close the Create GUI Checkpoint dialog before specifying arguments for the Date Format, Time Format, Range and RegularExpression property checks on the "Departure Time:" edit object in the sample Flights application:

Check	Object	Arguments
DateFormat	Agent Name:	---
TimeFormat	Agent Name:	---
Range	Agent Name:	---
RegularExpression	Agent Name:	---

Argument Specification

Please specify arguments for the following checks:

Specify Arguments...   Close   Help
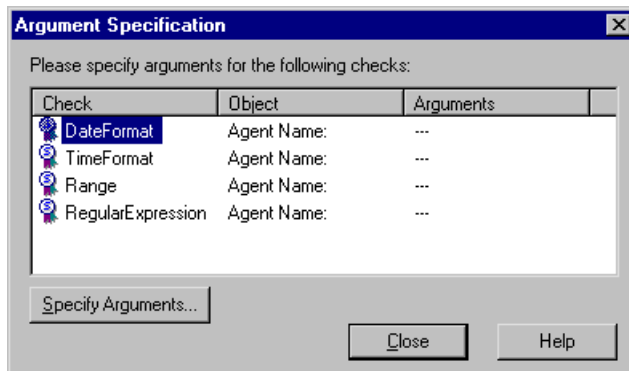
The *property check* appears in the **Check** column. The *logical name* of the object appears in the **Object** column. Three dots appear in the **Arguments** column to indicate that the arguments for the property check have not been specified.
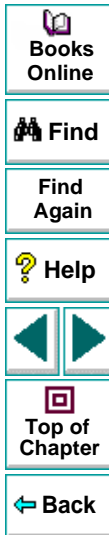
To specify arguments from the Argument Specification dialog box:

1 In the **Check** column, select a property check.

2 Click the **Specify Arguments** button. Alternatively, double-click the property check.

3 The dialog box for specifying arguments for that property check opens.

4 Specify the arguments for the property check, as described above.

5 Click **OK** to close the dialog box for specifying arguments.

6 Repeat the above steps until arguments appear in the **Arguments** column for all property checks.

7 Once arguments are specified for all property checks in the dialog box, click **Close** to close it and return to the GUI Checkpoint dialog box that is open.

8 Click **OK** to close the GUI Checkpoint dialog box that is open.

Books Online

Find

Find Again

Help

Top of Chapter

Back

Editing the Expected Value of a Property

When you create a GUI checkpoint, WinRunner captures the current property values for the objects you check. These current values are saved as *expected values* in the *expected results folder*.

When you run your test, WinRunner captures these property values again. It compares the new values captured during the test with the expected values that were stored in the test's expected results folder.

Suppose that you want to change the value of a property after it has been captured in a GUI checkpoint but before you run your test script. You can simply edit the expected value of this property in the Check GUI dialog box or the Create GUI Checkpoint dialog box.

Note that you cannot edit expected property values in the Edit GUI Checklist dialog box: When you open the Edit GUI Checklist dialog box, WinRunner does not capture current values. Therefore, this dialog box does not display expected values that can be edited.

Note: If you want to edit the expected value for a property check that is already part of a GUI checkpoint, you must change the expected results of the GUI checkpoint. For more information, see **Modifying the Expected Results of a GUI Checkpoint** on page 266.

To edit the expected value of an object:

1 If the Check GUI dialog box or the Create GUI Checkpoint dialog box is not already open, choose **Create > GUI Checkpoint > For Multiple Objects** to open the **Create GUI Checkpoint** dialog box and click **Open** to open the checklist in which to edit the expected value. Note that the Check GUI dialog box opens only when you create a new GUI checkpoint.

2 In the **Objects** pane, select an object.

3 In the **Properties** pane, select the property whose expected value you want to edit.

4 Do one of the following:

- Click the **Edit Expected Value** button.

- Double-click the existing expected value (the current value).

- Right-click with the mouse and choose **Edit Expected Value** from the pop-up menu.

Depending on the property, an edit field, an edit box, a list box, a spin box, or a new dialog box opens.

For example, when you edit the expected value of the **Enabled** property for a push_button class object, a list box opens:
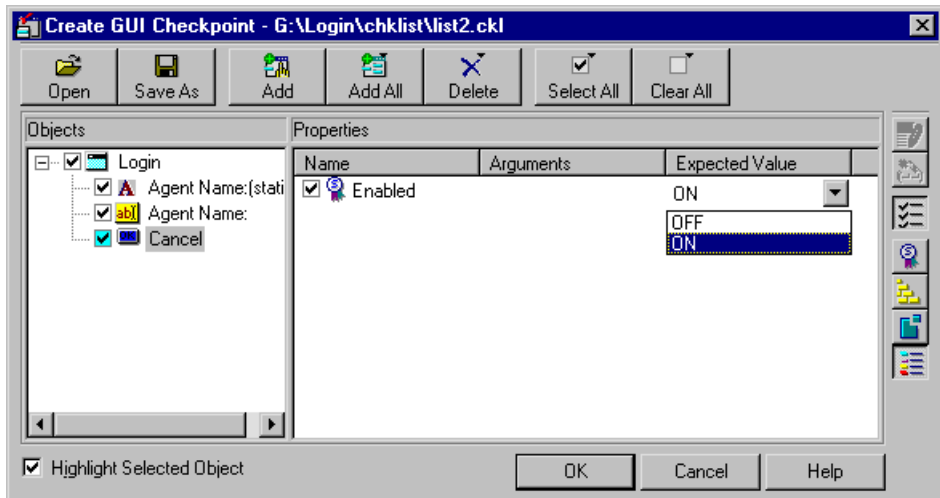
5 Edit the expected value of the property, as desired.

6 Click **OK** to close the dialog box.
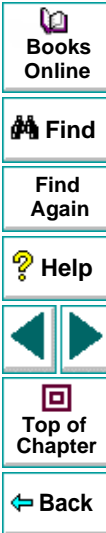
Modifying the Expected Results of a GUI Checkpoint

You can modify the expected results of an existing GUI checkpoint by changing the expected value of a property check within the checkpoint. You can make this change before or after you run your test script.

To modify the expected results for an existing GUI checkpoint:

1 Choose **Tools > Test Results** or click **Test Results**.

The WinRunner Test Results window opens.
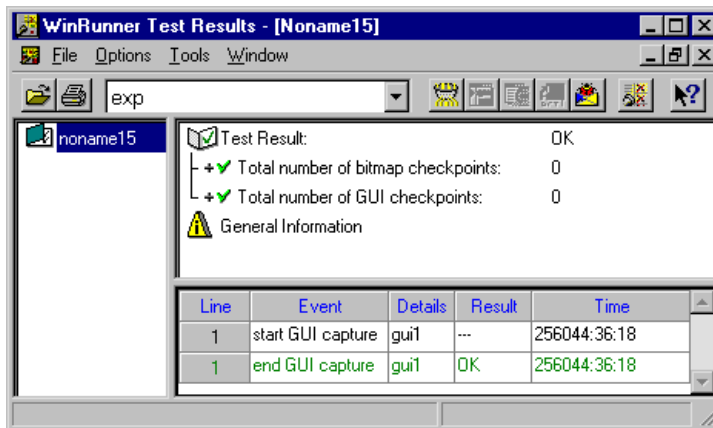
2 In the **Results** box, choose your expected results folder (by default, "exp").
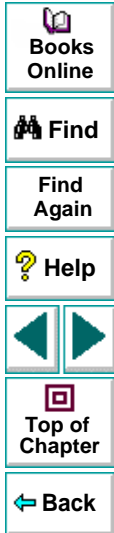
3 In the test log, locate the GUI checkpoint by looking for entries that list "end GUI capture" in the **Event** column. Note that the line number in the test script appears in the **Line** column of the test log.

> **Note:** You can use the **Show TSL** button to open the test script to the highlighted line number.

4 Double-click the desired "end GUI capture" entry, or click this entry and click **Display**.

The GUI Checkpoint Results dialog box opens.
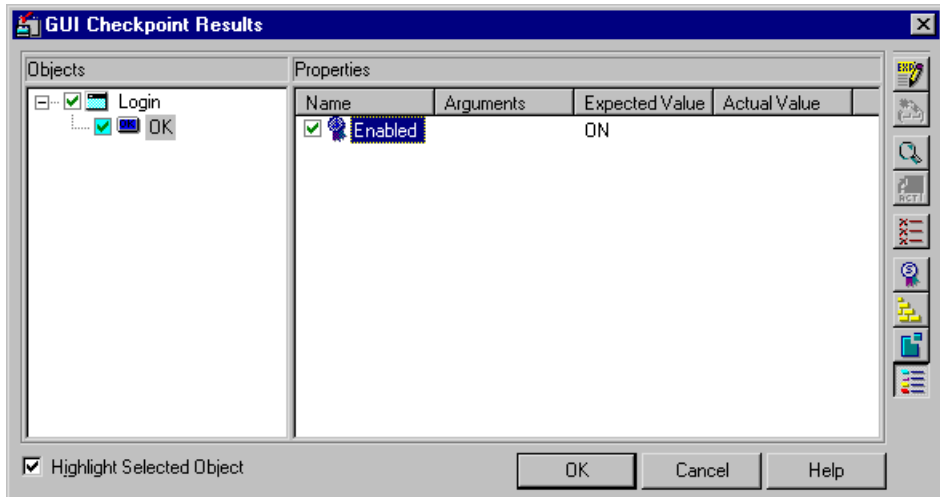
5 Select the property check whose expected results you want to modify. Click the
Edit expected value button. In the **Expected Value** column, modify the value,
as desired. Click OK to close the dialog box.

Note: You can also modify the expected value of a property check while creating a GUI checkpoint. For more information, see **Editing the Expected Value of a Property** on page 263.

Note: You can also modify the expected value of a GUI checkpoint to the actual value after a test run. For more information, see **Updating the Expected Results of a Checkpoint** on page 736.

Books Online

Find

Find Again

Help

Top of Chapter

Back

Creating Tests
Working with ActiveX and Visual Basic Controls

WinRunner supports Context Sensitive testing on ActiveX controls (also called OLE or OCX controls) and Visual Basic controls in Visual Basic applications.

This chapter describes:

- **Choosing Appropriate Support for Visual Basic Applications**
- **Activating an ActiveX Control Method**
- **Viewing ActiveX and Visual Basic Control Properties**
- **Getting and Setting the Values of ActiveX and Visual Basic Control Properties**
- **Checking Sub-Objects of ActiveX and Visual Basic Controls**
- **Using TSL Table Functions with ActiveX Controls**

Books Online

Find

Find Again

? Help

Top of Chapter

Back
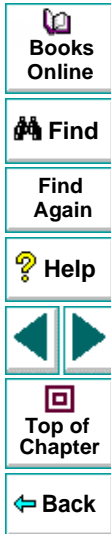
About Working with ActiveX and Visual Basic Controls

Many applications include ActiveX and Visual Basic controls developed by third-party organizations. WinRunner can record and run Context Sensitive operations on these controls, as well as check their properties.

WinRunner provides two types of support for ActiveX and Visual Basic controls within a Visual Basic application. You can either:

● compile a WinRunner agent into your application, and install and load add-in support for Visual Basic controls

● install and load add-in support for ActiveX and Visual Basic controls

When you work with the appropriate support, WinRunner recognizes ActiveX and Visual Basic controls, and treats them as it treats standard GUI objects. You can check the properties of ActiveX and Visual Basic controls as you check the properties of any standard GUI object. For more information, see Chapter 9, **Checking GUI Objects**.

At any time, you can view the current values of the properties of an ActiveX or a Visual Basic control using the ActiveX Properties Viewer. In addition, you can retrieve and set the values of properties for ActiveX controls using TSL functions. You can also use a TSL function to activate an ActiveX control method.

Books Online

M Find

Find Again

? Help

Top of Chapter

← Back

Note: You must start WinRunner before launching the application containing ActiveX controls.

WinRunner provides special built-in support for checking the contents or properties of ActiveX controls that are tables. For information on which TSL table functions are supported for specific ActiveX controls, see **Using TSL Table Functions with ActiveX Controls** on page 285. For information on checking the contents of an ActiveX table control, see Chapter 12, **Checking Table Contents**.

This chapter provides step-by-step instructions for checking ActiveX Control properties.

Choosing Appropriate Support for Visual Basic Applications

WinRunner provides two types of support for ActiveX and Visual Basic controls within a Visual Basic application. You can either:

● compile a WinRunner agent into your application, and install and load add-in support for Visual Basic controls

● install and load add-in support for ActiveX and Visual Basic controls

Before you test a Visual Basic application, it is best to add the WinRunner agent to your application, compile them together, and install the Visual Basic add-in from the WinRunner setup program, and load it from the Add-In Manager. If this is not possible, install and load both the ActiveX and Visual Basic add-ins from the WinRunner setup program, and load them both from the Add-In Manager.

The different levels of support are described below.

Books Online

Find

Find Again

Help

Top of Chapter

Back
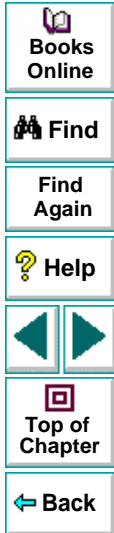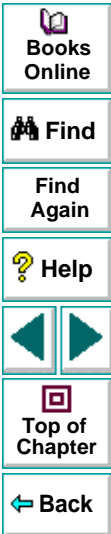
Working with the WinRunner Agent and Visual Basic Add-In Support

You can add a WinRunner agent, called *WinRunnerAddIn.Connect*, to your application and compile them together. The agent is in the *vbdev* folder on the WinRunner CD-ROM. For information on how to install and compile the agent, refer to the *readme.wri* file in the same folder. You can install add-in support for Visual Basic applications when you install WinRunner. For additional information, refer to your *WinRunner Installation Guide*. You can choose which installed add-ins to load for each session of WinRunner. For additional information, see **Loading WinRunner Add-Ins** on page 50.

When you add the WinRunner agent to your application and compile them together, you can:

● record and run tests with operations on ActiveX and standard Visual Basic controls

● uniquely identify names of internal ActiveX and Visual Basic controls

● create GUI checkpoints which check the properties of standard Visual Basic controls

● use the **ActiveX_get_info** and **ActiveX_set_info** TSL functions with ActiveX and Visual Basic controls

Books Online

M Find

Find Again

? Help

Top of Chapter

← Back
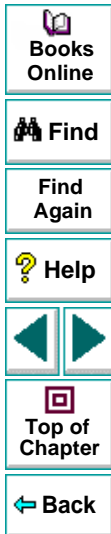
Working with ActiveX and Visual Basic Add-In Support without the WinRunner Agent

You can install add-in support for ActiveX and Visual Basic applications when you install WinRunner. For additional information, refer to your *WinRunner Installation Guide*. You can choose which installed add-ins to load for each session of WinRunner. For additional information, see **Loading WinRunner Add-Ins** on page 50.

When you install and load the ActiveX and Visual Basic add-ins without using the WinRunner agent, you can:

- record and run tests with operations on ActiveX and standard Visual Basic controls

- use the **ActiveX_get_info** and **ActiveX_set_info** TSL functions with ActiveX controls only

Books Online

Find

Find Again

Help

Top of Chapter

Back

Activating an ActiveX Control Method

You use the **ActiveX_activate_method** function to invoke an ActiveX method of an ActiveX control. You can insert this function into the test script using the Function Generator. The syntax of this function is:

ActiveX_activate_method (*object***,** *ActiveX_method***,** *return_value***,**
[*parameter1***,...,***parameter8***]);**

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

Viewing ActiveX and Visual Basic Control Properties

You use the ActiveX Properties Viewer to see the properties and property values for an ActiveX or Visual Basic control. You open the ActiveX Properties Viewer from the Tools menu. Note that you must load the ActiveX add-in in order to open the ActiveX Properties Viewer. You may also view ActiveX and Visual Basic control properties using the GUI checkpoint dialog boxes. For information on using the GUI checkpoint dialog boxes, see Chapter 9, **Checking GUI Objects**.
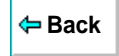
To view the properties of an ActiveX or a Visual Basic control:

1 Choose **Tools > ActiveX Properties Viewer** to open the ActiveX Properties Viewer.
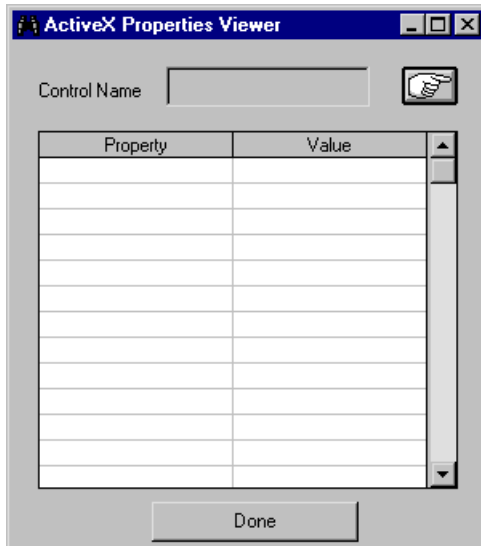
2 Click the pointing hand and click an ActiveX or Visual Basic control.

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

Property	Value
CalStatic	False
Caption	
Container	Object Reference -
DataChanged	False
DataField	
DataSource	Object Reference - 0
Date	3/10/99
DateArr	Array of BSTR
DateFormat	4
DateSelOn	0
Day	10
DayArr	Array of INTEGER
DayHeight	Array of LONG

Control Name: txtCalendar
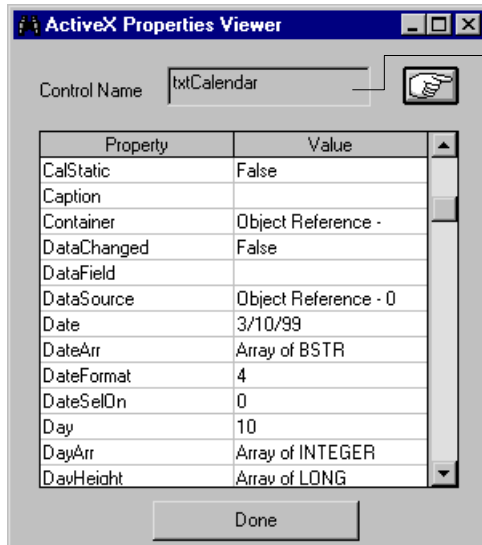
In this example, the control is a Visual Basic object in a Visual Basic application.

For ActiveX controls, the class name appears in this box.

Books Online

Find

Find Again

Help

Top of Chapter

Back

Note: When "Object Reference" appears in the Value column, it refers to the object's sub-objects and their properties. When "Array..." appears in the Value column, this indicates either an array of type or a two-dimensional array. You can use the **ActiveX_get_info** function to retrieve these values. For information on the **ActiveX_get_info** function, see **Getting the Value of an ActiveX or Visual Basic Control Property** on page 280 or refer to the *TSL Online Reference*.

4 Click **Done** to close the dialog box.

Getting and Setting the Values of ActiveX and Visual Basic Control Properties

Two TSL functions enable you to retrieve and set the values of properties for ActiveX and Visual Basic 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 20, **Generating Functions**.

Getting the Value of an ActiveX or Visual Basic Control Property

Use the **ActiveX_get_info** function to retrieve the value of any ActiveX or Visual Basic control property. The syntax of this function is:

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

Setting the Value of an ActiveX or Visual Basic Control Property

Use the **ActiveX_set_info** function to set the value for any ActiveX or Visual Basic control property. The syntax of this function is:

ActiveX_set_info (*object*, *property*, *out_value* **[,** *type***]);**

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

Checking Sub-Objects of ActiveX and Visual Basic Controls

ActiveX and Visual Basic controls may contain sub-objects, which contain their own properties. An example of a sub-object is Font. Note that Font is a sub-object because it cannot be highlighted in the application you are testing. When you load the appropriate add-in support, you can create a GUI checkpoint that checks the properties of a sub-object using the Check GUI dialog box. For information on GUI checkpoints, see Chapter 9, **Checking GUI Objects**.

In the example below, WinRunner checks the properties of the Font sub-object of an ActiveX table control. The example in the procedure below uses WinRunner with add-in support for Visual Basic and the Flights table in the sample Visual Basic Flights application.

To check the sub-objects of an ActiveX or a Visual Basic control:

 1 Choose **Create > GUI Checkpoint > For Object/Window** or click the **GUI Checkpoint for Object/Window** button on the User toolbar.

 2 Double-click the control in the application you are testing.

WinRunner may take a few seconds to capture information about the control.

Books Online

Find

Find Again

Help

Top of Chapter

Back

The **Check GUI** dialog box opens.
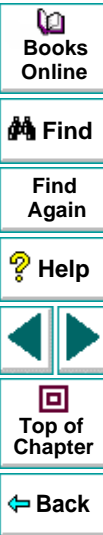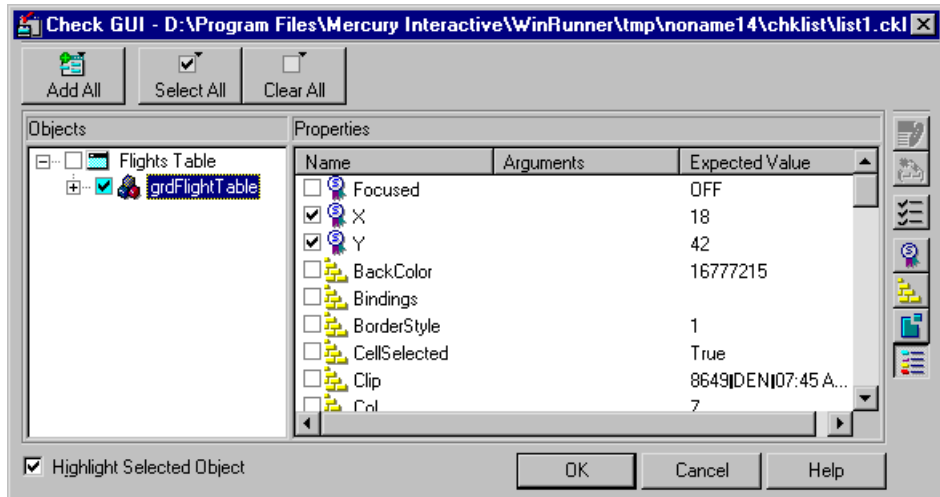
3 In the **Objects** pane, click the Expand sign (+) beside the object to display its sub-objects, and select a sub-object to display its ActiveX control properties.
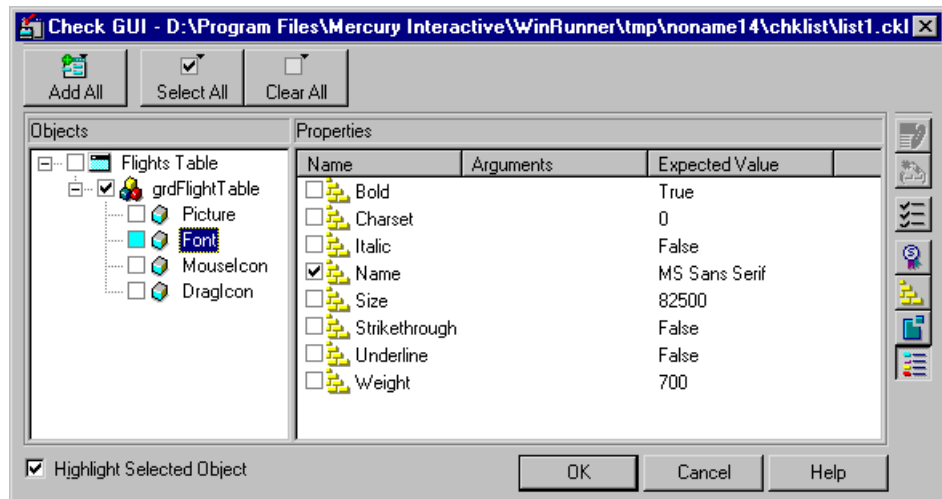
The **Objects** pane displays the object and its sub-objects. In this example, the sub-objects are displayed under the "grdFlightTable" object. The **Properties** pane displays the properties of the sub-object that is highlighted in the Objects pane. Note that each sub-object has one or more default property checks. In this example, the properties of the Font sub-object are displayed, and the Name property of the Font sub-object is selected as a default check.
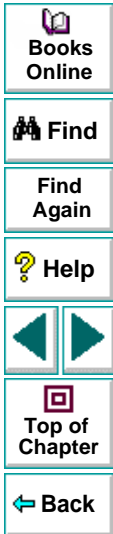
Specify which sub-objects of the table to check: first, select a sub-object in the Objects pane; next, select the properties to check in the Properties pane.

Note that since this ActiveX control is a table, by default, checks are selected on the Height, Width, and Table Content properties. If you do not want to perform these checks, clear the appropriate check boxes. For information on checking table contents, see Chapter 12, **Checking Table Contents**.
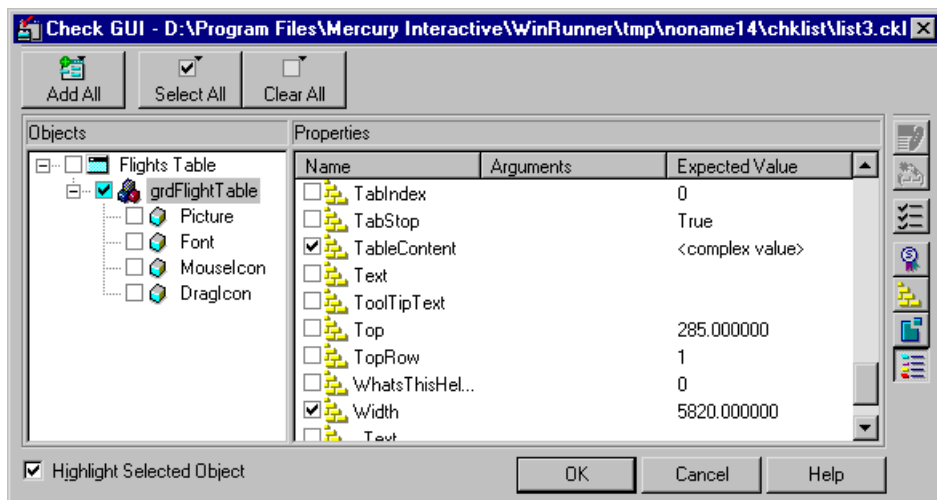
4 Click **OK** to close the dialog box.

An **obj_check_gui** statement is inserted into your test script. For more information on the **obj_check_gui** function, see Chapter 9, **Checking GUI Objects**, or refer to the *TSL Online Reference*.
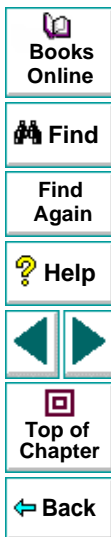
Using TSL Table Functions with ActiveX Controls

You can use the TSL **tbl_** functions to work with a number of ActiveX controls. WinRunner contains built-in support for the ActiveX controls and the functions in the table below. For detailed information about each function and examples of usage, refer to the *TSL Online Reference*.

Books Online

Find

Find Again

Help

Top of Chapter

Back
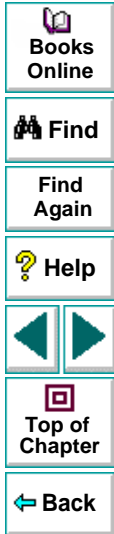
	Data Bound Grid Control	FarPoint Spreadsheet Control	MicroHelp MH3d List Control	Microsoft Grid Control	Sheridan Data Grid Control	True DBGrid Control
tbl_activate_cell	✔	✔	✔	✔	✔	✔
tbl_activate_header	✔	✔	✔	✔	✔	✔
tbl_get_cell_data	✔	✔	✔	✔	✔	✔
tbl_get_cols_count	✔	✔	✔	✔	✔	✔
tbl_get_column_name	✔	✔	✔	✔	✔	✔
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	✔	✔	✔	✔		✔

Books Online

Find

Find Again

Help

Top of Chapter

Back

Creating Tests
Checking PowerBuilder Applications

When you work with WinRunner with added support for PowerBuilder applications, you can create GUI checkpoints to check PowerBuilder objects in your application.

This chapter describes:

- **Checking Properties of DropDown Objects**
- **Checking Properties of DataWindows**
- **Checking Properties of Objects within DataWindows**
- **Working with Computed Columns in DataWindows**

Books Online

Find

Find Again

Help

Top of Chapter

Back

About Checking PowerBuilder Applications

You can use GUI checkpoints to check the *properties* of PowerBuilder objects in your application. When you check these properties, you can check the *contents* of PowerBuilder objects as well as their standard GUI properties. This chapter provides step-by-step instructions for checking the properties of the following PowerBuilder objects:

- DropDown objects

- DataWindows

- DataWindow columns

- DataWindow text

- DataWindow reports

- DataWindow graphs

- computed columns in a DataWindow

Books Online

Find

Find Again

Help

Top of Chapter

Back

Checking Properties of DropDown Objects

You can create a GUI checkpoint that checks the properties, including contents, of a DropDown list or a DropDown DataWindow. You can check the same properties, including contents, for a DropDown DataWindow that you can check for a regular DataWindow. Note that before creating a GUI checkpoint on a DropDown object, you should first record a **tbl_set_selected_cell** statement in your test script. Use the CHECK GUI FOR OBJECT/WINDOW softkey to create the GUI checkpoint while recording. You create a GUI checkpoint that checks the contents of a DropDown object as you would create one for a table. For information on checking tables, see Chapter 12, **Checking Table Contents**.

Checking Properties of a DropDown Object with Default Checks

You can create a GUI checkpoint that performs a default check on a DropDown object. A default check on a DropDown object includes a case-sensitive check on the contents of the entire object. WinRunner uses column names and the index number of rows to check the cells in the object.

You can also perform a check on a DropDown object in which you specify which checks to perform. For additional information, see **Checking Properties of a DropDown Object while Specifying which Checks to Perform** on page 291.

To check the properties of a DropDown object with default checks:

1 Choose **Create > Record–Context Sensitive** or click the **Record–Context Sensitive** button.

2 Click in the DropDown object to record a **tbl_set_selected_cell** statement in your test script

3 While recording, press the CHECK GUI FOR OBJECT/WINDOW softkey.

4 Click in the DropDown object once.

WinRunner captures the GUI information and stores it in the test's expected results folder. 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 Properties of a DropDown Object while Specifying which Checks to Perform

You can create a GUI checkpoint in which you specify which checks to perform on a DropDown object. When you double-click in a DropDown object while creating a GUI checkpoint, the Check GUI dialog box opens. For example, if you are checking a DropDownListBox, you double-click the **DropDownListBoxContent** property check in the Check GUI dialog box to open the Edit Check dialog box. In the Edit Check dialog box, you can specify the scope of the content check on the object, select the verification types and method, and edit the expected value of the DataWindow contents.

To check the properties of a DropDown object while specifying which checks to perform:

1 Choose **Create > Record–Context Sensitive** or click the **Record–Context Sensitive** button.

2 Click in the DropDown object to record a **tbl_set_selected_cell** statement in your test script.

3 While recording, press the CHECK GUI FOR OBJECT/WINDOW softkey.

4 Double-click in the DropDown object.
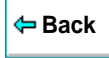
The Check GUI dialog box opens.
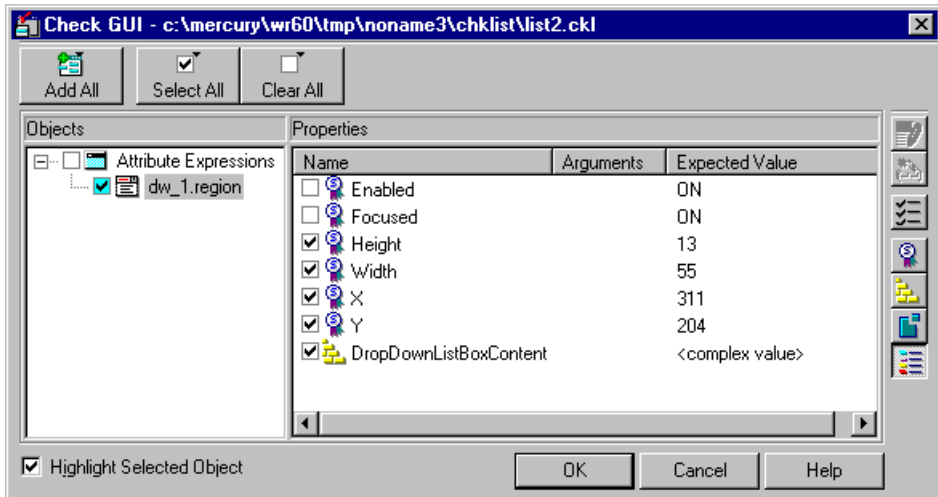
The example above displays the Check GUI dialog box for a DropDown list. The Check GUI dialog box for a DropDown DataWindow is identical to the dialog box for a DataWindow.

5 In the **Properties** pane, select the **DropDownListBoxContent** check and click the **Edit Expected Value** button, or double-click the "<complex value>" entry in the **Expected Value** column.

The **Edit Check** dialog box opens.

6 You can select which checks to perform and edit the expected data. For additional information on using this dialog box, see **Understanding the Edit Check Dialog Box** on page 312.

7 When you are done, click **OK** to save your changes, close the Edit Check dialog box, and restore the Check GUI dialog box.

8 Click **OK** to close the Check GUI dialog box.

WinRunner captures the GUI information and stores it in the test's expected results folder. 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.*

Note: If you wish to check additional objects while performing a check on the contents, use the **Create > GUI Checkpoint > For Multiple Objects** command (instead of the **Create > GUI Checkpoint > For Object/Window** command), which inserts a **win_check_gui** statement into your test script. For information on checking the standard GUI properties of DropDown objects, see Chapter 9, **Checking GUI Objects**.

Checking Properties of DataWindows

You can create a GUI checkpoint that checks the properties of a DataWindow. One of the properties you can check is **DWTableContent**, which is a check on the contents of the DataWindow. You create a content check on a DataWindow as you would create one on a table. For additional information on checking table contents, see Chapter 12, **Checking Table Contents**.

Checking Properties of a DataWindow with Default Checks

You can create a GUI checkpoint that checks the properties of a DataWindow with default checks. There are different default checks for different types of DataWindows.

To check the properties of a DataWindow with default checks:

1 Choose **Create > GUI Checkpoint > For Object/Window** or click the **GUI Checkpoint for Object/Window** button on the User toolbar.

2 Click in the DataWindow once.

WinRunner captures the GUI information and stores it in the test's expected results folder. 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 Properties of a DataWindow while Specifying which Checks to Perform

You can create a GUI checkpoint that checks the properties of a DataWindow while specifying which checks to perform.

**To check the properties of a DataWindow while specifying which checks to perform:**

1 Choose **Create > GUI Checkpoint > For Object/Window** or click the **GUI Checkpoint for Object/Window** button on the User toolbar.

2 Double-click in the DataWindow.

3 The Check GUI dialog box opens.
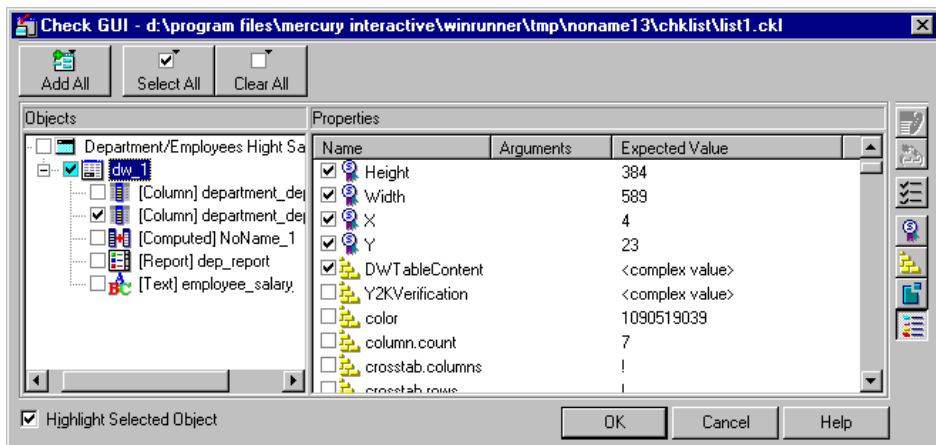
Note that the properties of objects within a DataWindow are displayed in the dialog box. WinRunner can perform checks on these objects. For additional information, see **Checking Properties of Objects within DataWindows** on page 297.

4 Select the **DWTableContent** check and click the **Edit Expected Value** button, or double-click the "<complex value>" entry in the **Expected Value** column.

The **Edit Check** dialog box opens.

5 You can select which checks to perform and edit the expected data. For additional information on using this dialog box, see **Understanding the Edit Check Dialog Box** on page 312.

6 When you are done, click **OK** to save your changes, close the Edit Check dialog box, and restore the Check GUI dialog box.

7 Click **OK** to close the Check GUI dialog box.

WinRunner captures the GUI information and stores it in the test's expected results folder. 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*.

Books Online

Find

Find Again

Help

Top of Chapter

Back

Checking Properties of Objects within DataWindows

You can create a GUI checkpoint that checks the properties of the following DataWindow objects:

- DataWindows

- DataWindow columns

- DataWindow text

- DataWindow reports

- DataWindow graphs

- DataWindow computed columns

DataWindow objects cannot be highlighted in the application you are testing. You can create a GUI checkpoint that checks the properties of objects within DataWindows using the Check GUI dialog box. For information on GUI checkpoints, see Chapter 9, **Checking GUI Objects**.

To check the properties of objects in a DataWindow:

 1 Choose **Create > GUI Checkpoint > For Object/Window** or click the **GUI Checkpoint for Object/Window** button on the User toolbar.

 2 Double-click the DataWindow in the application you are testing.

WinRunner may take a few seconds to capture information about the DataWindow.

The **Check GUI** dialog box opens.
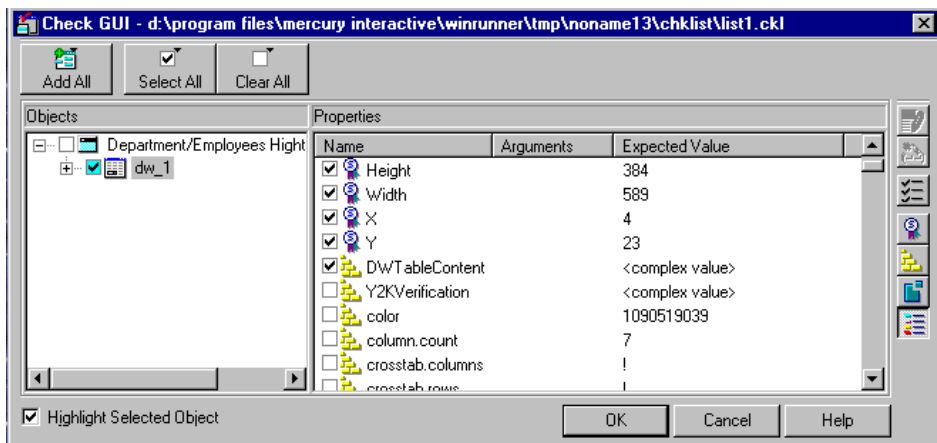
3 In the **Objects** pane, click the Expand sign (**+**) beside the DataWindow to display its objects, and select an object to display its properties.
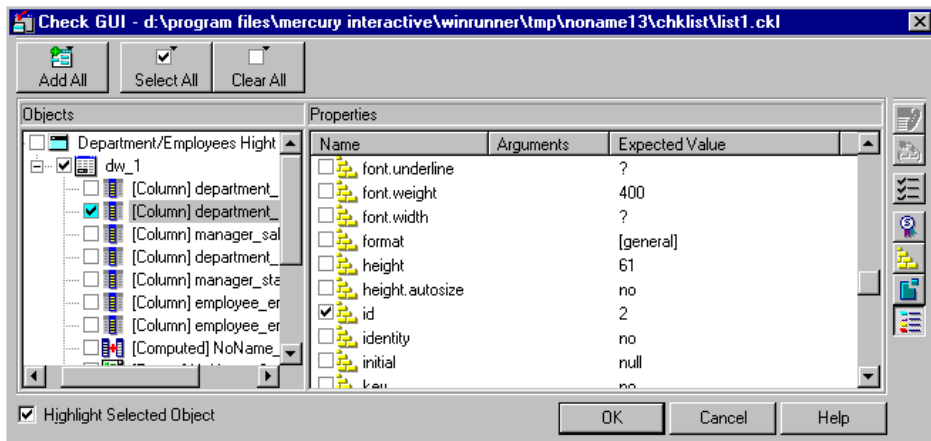
The **Objects** pane displays the DataWindow and the objects within it. The **Properties** pane displays the properties of the object in the DataWindow that is highlighted in the Objects pane. These objects can be columns, computed columns, text, graphs, and reports. Note that each object has one or more default property checks.

Specify which objects of the DataWindow to check: first, select an object in the Objects pane; next, select the properties to check in the Properties pane.

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

An **obj_check_gui** statement is inserted into your test script. For more information on the **obj_check_gui** function, see Chapter 9, **Checking GUI Objects**, or refer to the *TSL Online Reference*.

Books Online

Find

Find Again

Help
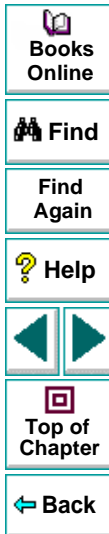
Note: If an object in a DataWindow is displayed in the Objects pane of the GUI checkpoint dialog boxes as "NoName," then the object has no internal name.

Top of Chapter

Back

Working with Computed Columns in DataWindows

If computed columns are placed in detail band of the DataWindow, WinRunner can record and run tests on them. WinRunner uses the **tbl_get_selected_cell**, **tbl_activate_cell**, and **tbl_get_cell_data** TSL functions to record and run tests on computed columns. For information on using these TSL functions, refer to the *TSL Online Reference*.

WinRunner can also retrieve data about computed columns which are not placed in detail band of the DataWindow, using the **tbl_get_cell_data** TSL function. For information about this TSL function, refer to the *TSL Online Reference*.

To check the contents of computed columns in detail band of the DataWindow, use the **DWComputedContent** property check.

You cannot refer to a computed column by its index, since the computed column is not part of the database. Therefore, you must refer to a computed column by its name.

- Record a selection on the computed column. The name of the column appears in the **tbl_selected_cell** statement inserted in your test script.

- Perform a GUI checkpoint on the DataWindow in which the computed column appears. The name of the computed column appears in the Objects pane below the name of the parent DataWindow.

When you work with WinRunner with added support for application development environments such as Visual Basic, PowerBuilder, Delphi, and Oracle, you can create GUI checkpoints that check the contents of tables in your application.

This chapter describes:

- **Checking Table Contents with Default Checks**
- **Checking Table Contents while Specifying Checks**
- **Understanding the Edit Check Dialog Box**

About Checking Table Contents

Tables are generally part of a specific development environment application, such as Visual Basic, PowerBuilder, Delphi, and Oracle. These toolkits can display database information in a grid. In order to perform the checks on a table described in this chapter, you must install and load add-in support for the relevant development environment. You can choose to install support for Visual Basic or PowerBuilder applications when you install WinRunner. In addition, you can install support for other development environments, such as Delphi and Oracle, separately. You can use the Add-In Manager dialog box to choose which add-in support to load for each session of WinRunner. For information on the Add-In Manager dialog box, see Chapter 2, **WinRunner at a Glance**. For information on displaying the Add-In Manager dialog box, see Chapter 36, **Setting Global Testing Options**.

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

Books Online

Find

Find Again

Help

Top of Chapter

Back

You can create a GUI checkpoint for table contents by clicking in the table 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** or a **win_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. You can view the results of the checkpoint in the WinRunner Test Results Window. For more information, see Chapter 28, **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 the contents of tables.

You can also create a GUI checkpoint that checks the contents of a PowerBuilder DropDown list or a DataWindow: you check a DropDown list as you would check a single-column table; you check a DataWindow as you would check a multiple-column table. For additional information, see Chapter 11, **Checking PowerBuilder Applications**.

In addition to checking a table's contents, you can also check its other properties. If a table contains ActiveX properties, you can check them in a GUI checkpoint. WinRunner also has built-in support for ActiveX controls that are tables. For additional information, see Chapter 10, **Working with ActiveX and Visual Basic Controls**. You can also check a table's standard GUI properties in a GUI checkpoint. For additional information, see Chapter 9, **Checking GUI Objects**.

Checking Table Contents with Default Checks

You can create a GUI checkpoint that performs a default check on the contents of a table.

A default check performs a case-sensitive check on the contents of the entire table. WinRunner uses column names and the index number of rows to locate the cells in the table.

You can also perform a check on table contents in which you specify which checks to perform. For additional information, see **Checking Table Contents while Specifying Checks** on page 308.

To check table contents with a default check:

1 Choose **Create > GUI Checkpoint > For Object/Window** or click the **GUI Checkpoint for Object/Window** button on the User toolbar.

2 Click in the table in the application you are testing.

WinRunner may take a few seconds to capture information about the table.

An **obj_check_gui** statement is inserted into your test script. For more information on the **obj_check_gui** function, refer to the *TSL Online Reference*.

Books Online

Find

Find Again

Help

Top of Chapter

Back

Note: If you wish to check other table object properties while performing a check on the table contents, use the **Create > GUI Checkpoint > For Multiple Objects** command (instead of the **Create > GUI Checkpoint > For Object/Window** command), which inserts a **win_check_gui** statement into your test script. For information on checking the standard GUI properties of tables, see Chapter 9, **Checking GUI Objects**. For information on checking the ActiveX control properties of a tables, see Chapter 10, **Working with ActiveX and Visual Basic Controls**.

Checking Table Contents while Specifying Checks

You can use a GUI checkpoint to specify which checks to perform on the contents of a table. To create a GUI checkpoint on table contents in which you specify checks, you choose a GUI checkpoint command and double-click in the table.

The example in the procedure below uses WinRunner with add-in support for Visual Basic and the Flights table in the sample Visual Basic Flights application.

To check table contents while specifying which checks to perform:

1 Choose **Create > GUI Checkpoint > For Object/Window** or click the **GUI Checkpoint for Object/Window** button on the User toolbar.

2 Double-click in the table in the application you are testing.

Books Online

Find

Find Again

Help

Top of Chapter

Back
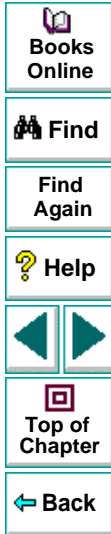
WinRunner may take a few seconds to capture information about the table, and then the **Check GUI** dialog box opens.
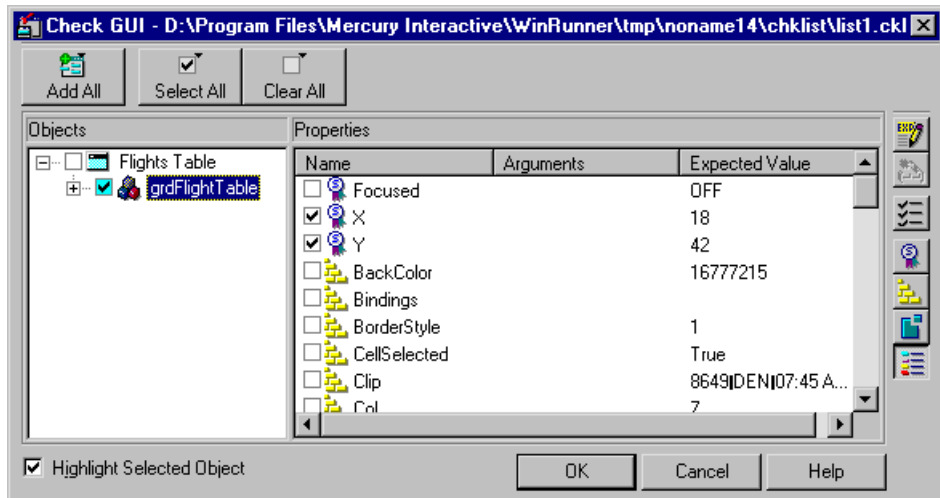
The dialog box displays the table's unique table properties as nonstandard objects.

3 Scroll down in the dialog box or resize it so that the **TableContent** property check is displayed in the **Properties** pane. Note that the table contents property check may have a different name than **TableContent**, depending on which toolkit is used.
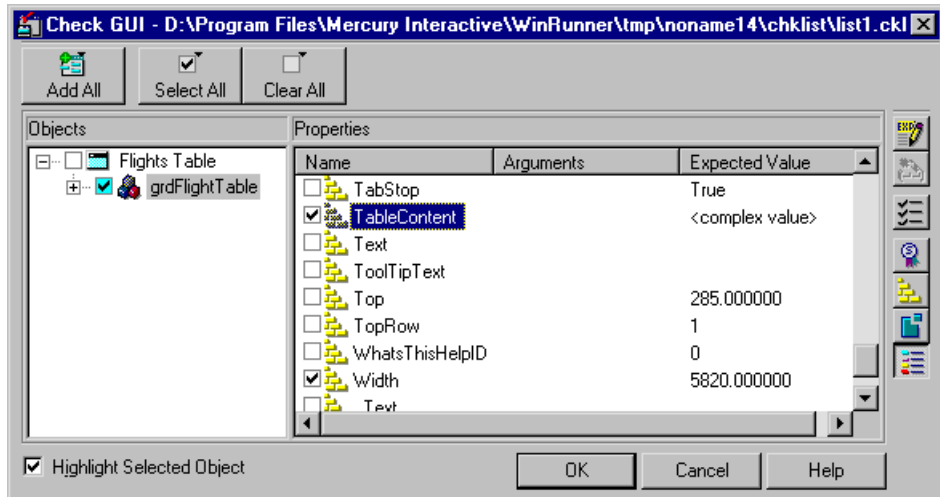
4 Select the **TableContent** (or corresponding) property check and click the **Edit Expected Value** button. Note that <complex value> appears in the Expected Value column for this property check, since the expected value of this check is too complex to be displayed in this column.

The **Edit Check** dialog box opens.

5 You can select which cells to check and edit the expected data. For additional information on using this dialog box, see **Understanding the Edit Check Dialog Box** on page 312.

6 When you are done, click **OK** to save your changes, close the Edit Check dialog box, and restore the Check GUI dialog box.

7 Click **OK** to close the Check GUI dialog box.

An **obj_check_gui** statement is inserted into your test script. For more information on the **obj_check_gui** function, refer to the *TSL Online Reference.*

Note: If you wish to check other table object properties while performing a check on the table contents, use the **Create > GUI Checkpoint > For Multiple Objects** command (instead of the **Create > GUI Checkpoint > For Object/Window** command), which inserts a **win_check_gui** statement into your test script. For information on checking the standard GUI properties of tables, see Chapter 9, **Checking GUI Objects**. For information on checking the ActiveX control properties of a tables, see Chapter 10, **Working with ActiveX and Visual Basic Controls**.

Books Online

Find

Find Again

Help

Top of Chapter

Back

Understanding the Edit Check Dialog Box

The **Edit Check** dialog box enables you to specify which cells in a table to check, and which verification method and verification type to use. You can also edit the expected data for the table cells included in the check.
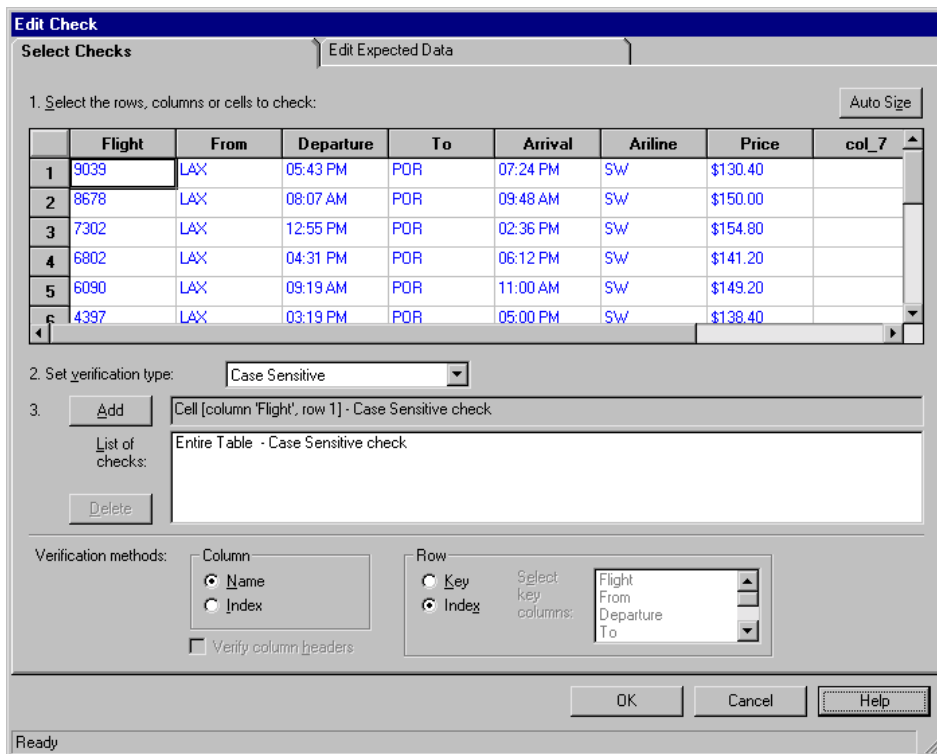
In the **Select Checks** tab, you can specify the information that is saved in the GUI checklist:

- which table cells to check

- the verification method

- the verification type

Note that if you are creating a check on a single-column table, the contents of the Select Checks tab of the Edit Check dialog box differ from what is shown above. For additional information, see **Specifying the Verification Method for a Single-Column Table** on page 317.

Specifying which Cells to Check

The **List of Checks** pane displays all the checks that will be performed, including the verification type. When the Edit Check dialog box is opened for the first time for a checkpoint, the default check is displayed:

- The default check for a multiple-column table is a case sensitive check on the entire table by column name and row index.

- The default check for a single-column table is a case sensitive check on the entire table by row position.

Note: If your table contains multiple columns with the same name, WinRunner disregards the duplicate columns and does not perform checks on them. Therefore, you should select the column index option.

If you do not wish to accept the default settings, you must delete the default check before you specify the checks to perform. Select the "Entire Table - Case Sensitive check" entry in the **List of Checks** box and click the **Delete** button. Alternatively, double-click this entry in the **List of Checks** box. A WinRunner message prompts you to delete the highlighted check. Click **Yes**.

Books Online

M Find

Find Again

? Help

Top of Chapter

Back

Next, specify the checks to perform. You can choose different verification type for different selections of cells. Therefore, specify the verification type before selecting cells. For more information, see **Specifying the Verification Type** on page 319.

Highlight the cells on which you want to perform the content check. Next, click the **Add** button toolbar to add a check for these cells. Alternatively, you can:

- double-click a cell to check it

- double-click a row header to check all the cells in a row

- double-click a column header to check all the cells in a column

- double-click the top-left corner to check the entire table

A description of the cells to be checked appears in the **List of Checks** box.

Specifying the Verification Method

You can select the verification method to control how WinRunner identifies columns or rows within a table. The verification method applies to the entire table. Specifying the verification method is different for multiple-column and single-column tables.

Specifying the Verification Method for a Multiple-Column Table

Column

- **Name:** 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.

- **Index:** WinRunner looks for the selection according to the index, or position, of the columns. A shift in the position of the columns within the table results in a mismatch. Select this option if your table contains multiple columns with the same name. For additional information, see the note on **page 314**. Choosing this option enables the **Verify column headers** check box, which enables you to check column headers as well as cells.

Row

- **Key:** WinRunner looks for the rows in the selection according to the key(s) specified in the **Select key columns** list box, which lists the names of all columns in the table. A shift in the position of any of the rows does not result in a mismatch. If the key selection does not identify a unique row, only the first matching row will be checked.

- **Index** (default setting): WinRunner looks for the selection according to the index, or position, of the rows. A shift in the position of any of the rows results in a mismatch.
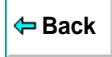
Specifying the Verification Method for a Single-Column Table

The Verification Method box in the Select Checks tab for a single-column table is different from that for a multiple-column table. The default check for a single-column table is a case sensitive check on the entire table by row position.

Books Online

Find

Find Again

Help

Top of Chapter

Back
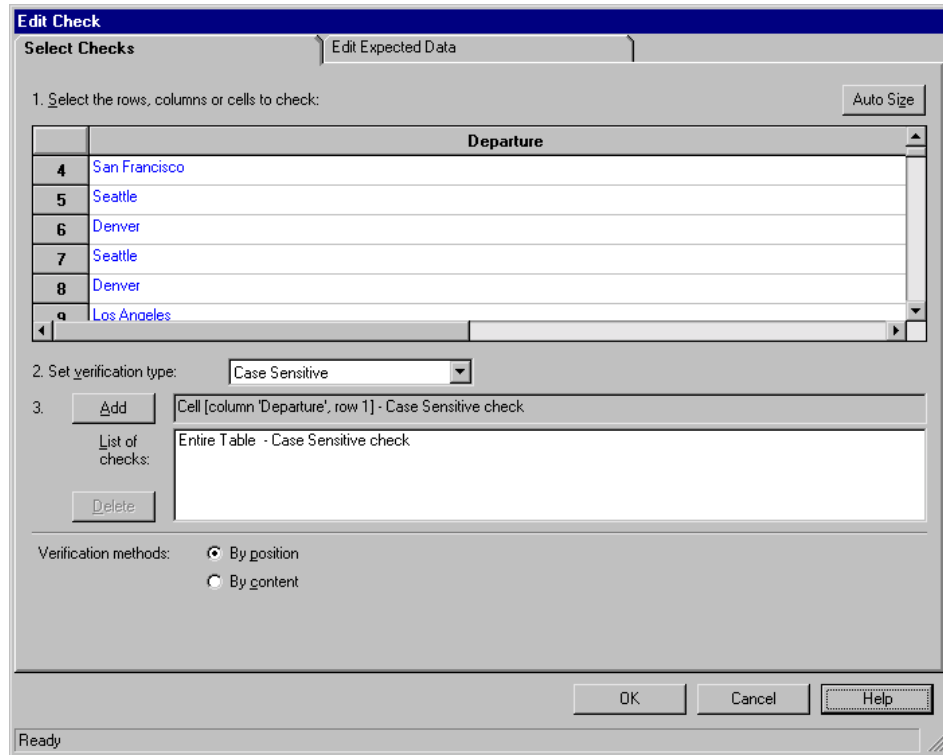
- **By position:** WinRunner checks the selection according to the location of the items within the column.

- **By content:** WinRunner checks the selection according to the content of the items, ignoring their location in the column.

Specifying the Verification Type

WinRunner can verify the contents of a table in several different ways. You can choose different verification types for different selections of cells.

- **Case Sensitive** (the default): WinRunner compares the text content of the selection. 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. 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.

- **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.

- **Case Sensitive Ignore Spaces:** WinRunner checks the data in the cell according to case and content, ignoring differences in spaces. WinRunner reports any differences in case or content as a mismatch.

- **Case Insensitive Ignore Spaces:** WinRunner checks the content in the cell according to content, ignoring differences in case and spaces. WinRunner reports only differences in content as a mismatch.

Click **OK** to save your changes to both tabs of the Edit Check dialog box. The dialog box closes and the Check GUI dialog box is restored.

Editing the Expected Data

To edit the expected data in the table, click the **Edit Expected Data** tab. If you previously saved changes in the Select Checks tab, you can click **Reload Table** to reload the table selections from the checklist. A WinRunner message prompts you to reload the saved data. Click **Yes**.

Note that if you previously saved changes to the Select Checks tab, and then reopened the Edit Check dialog box, the table appears color coded in the Edit Expected Data tab. The cells included in the check appear in blue on a white background. The cells excluded from the check appear in green on a yellow background.

To edit the expected value of data in a cell, double-click inside the cell. A cursor appears in the cell. Change the contents of the cell, as desired. Click **OK** to save your changes to both tabs of the Edit Check dialog box. The dialog box closes and the Check GUI dialog box is restored.

Creating Tests
Checking Databases

By adding database checkpoints to your test scripts, you can check the contents of databases in different versions of your application.

This chapter describes:

- **Choosing a Database**
- **Creating a Default Check on a Database**
- **Creating a Custom Check on a Database**
- **Messages in the Database Checkpoint Dialog Boxes**
- **Working with the Database Checkpoint Wizard**
- **Understanding the Edit Check Dialog Box**
- **Modifying a Database Checkpoint**
- **Modifying the Expected Results of a Database Checkpoint**
- **Parameterizing Database Checkpoints**
- **Using TSL Functions to Work with a Database**

Books Online

Find

Find Again

? Help

Top of Chapter

Back

About Checking Databases

You can use database checkpoints in your test script to check databases in your application and detect defects. You define a query on your database, and then you create a database checkpoint that checks the *properties* of the results of the query. When you check these properties, you can check the contents of the results or how many rows or columns the results contains.

There are three ways to create a database checkpoint:

- You can use Microsoft Query to create a *query* on a database. The results of a query on a database are known as a *result set.*You can install Microsoft Query from the *custom installation* of Microsoft Office.

- You can define an ODBC query manually, by creating its SQL statement.

- You can use Data Junction to create a *conversion* file that converts a database to a *target* text file. Note that Data Junction is not automatically included in your WinRunner package. To purchase Data Junction, contact your Mercury Interactive representative. For detailed information on working with Data Junction, refer to the documentation in the Data Junction package.

When you run your test, the database checkpoint compares the current values of the properties of the result set or target file to the expected results. If the expected results and the current results do not match, the database checkpoint fails.

Books Online

Find

Find Again

Help

Top of Chapter

Back

For purposes of simplicity, this chapter will refer to the result of the ODBC query or the target of the Data Junction conversion as a result set.

In WinRunner, you create a database checkpoint based on the results of the query you defined on a database. A database checkpoint can be either a default check or a custom check, in which you specify which properties to check.

You can use a default check to check the entire contents of a result set, or you can use a custom check to check the partial contents, the number of rows, and the number of columns of a result set. Information about which result set properties to check is saved in a *checklist*. WinRunner captures the current information about the database and saves this information as *expected results*. A *database checkpoint* is automatically inserted into the test script. This checkpoint appears in your test script as a **db_check** statement.

Books Online

M Find

Find Again

? Help

Top of Chapter

⬅ Back

For example, when you check the database of an application for the first time in a test script, the following statement is generated:

db_check("list1.cdl", "dbvf1");

where list1.cdl is the name of the *checklist* containing information about the database and the properties to check, and dbvf1 is the name of the *expected results file*. The checklist is stored in the test's *chklist* folder. If you are working with Microsoft Query or ODBC, it references a **.sql* query file, which contains information about the database and the SQL statement. If you are working with Data Junction, it references a **.djs* conversion file, which contains information about the database and the conversion. When you define a query, WinRunner creates a checklist and stores it in the test's *chklist* folder. The expected results file is stored in the test's *exp* folder. For more information on the **db_check** function, refer to the *TSL Online Reference*.

When you run the test, the database checkpoint compares the current state of the database in the application being tested to the expected results. If the expected results and the current results do not match, the database checkpoint fails. The results of the checkpoint can be viewed in the Test Results window. For more information, see Chapter 28, **Analyzing Test Results**.

Books Online

Find

Find Again

Help

Top of Chapter

Back

You can modify the expected results of an existing database checkpoint before or after you run your test. You can also make changes to the query in an existing database checkpoint. This is useful if you move the database to a new location on the network.

When you create a database checkpoint using ODBC/Microsoft Query, you can add parameters to an SQL statement to parameterize your checkpoint. This is useful if you want to create a database checkpoint on a query in which the SQL statement defining your query changes.
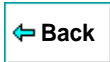
Choosing a Database

Before you can choose which database to check, you must start creating a database checkpoint. You can create either a default or a custom database checkpoint. For additional information, see **Creating a Default Check on a Database** on page 332 and **Creating a Custom Check on a Database** on page 337. While you are creating a checkpoint, you must specify which database to check. You can use the following tools to specify which database to check:

• ODBC/Microsoft Query

• Data Junction

Creating a Query in ODBC/Microsoft Query

You can use Microsoft Query to choose a data source and define a query within the data source, or you can define a connection string and an SQL statement manually. WinRunner supports the following versions of Microsoft Query:

• version 2.00 (in Microsoft Office 95)

• version 8.00 (in Microsoft Office 97)

• version 2000 (in of Microsoft Office 2000)

To creating a query in ODBC without using Microsoft Query, specify the connection string and the SQL statement in the Database Checkpoint wizard. For additional information, see **Specifying an SQL Statement** on page 352.

To choose a data source and define a query in Microsoft Query:

 1 Choose a new or an existing data source.

 2 Define a query.

Note: If you want to parameterize the SQL statement in the **db_check** statement which will be generated, then in the last wizard screen in Microsoft Query 8.00, click **View data or edit query in Microsoft Query**. Follow the instructions in **Guidelines for Parameterizing SQL Statements** on page 394.

---

 3 When you are done:

 ● In version 2.00, choose **File > Exit and return to WinRunner** to close Microsoft Query and return to WinRunner.

 ● In version 8.00, in the Finish screen of the Query Wizard, click **Exit and return to WinRunner** and click **Finish** to exit Microsoft Query. Alternatively, click **View data or edit query in Microsoft Query** and click **Finish**. After viewing or editing the data, choose **File > Exit and return to WinRunner** to close Microsoft Query and return to WinRunner.

4 Continue creating a database checkpoint in WinRunner:

● To create a default check on a database, follow the instructions starting at step 5 on **page 334**.

● To create a custom check on a database, follow the instructions starting at step 5 on **page 339**.

For additional information on working with Microsoft Query, refer to the Microsoft Query documentation.

Books Online

Find

Find Again

Help

Top of Chapter

Back

Creating a Conversion File in Data Junction

You can use Data Junction to create a conversion file which converts a database to a target text file. WinRunner supports versions 6.5 and 7.0 of Data Junction.

To create a conversion file in Data Junction:

1 Specify and connect to the source database.

2 Select an ASCII (delimited) target spoke type and specify and connect to the target file. Choose the "Replace File/Table" output mode.

3 Map the source file to the target file.

Note: If you are working with Data Junction version 7.0 and your source database includes values with delimiters (CR, LF, tab), then in the Target Properties dialog box, you must specify "\r\n\t" as the value for the **TransliterationIn** property. The value for the **TransliterationOut** property must be blank.

---

4 When you are done, click **File > Export Conversion** to export the conversion to a *.djs* conversion file.

5 The Database Checkpoint wizard opens to the **Select conversion file** screen. Follow the instructions in **Selecting a Data Junction Conversion File** on page 357.

Books
Online

M Find

Find Again

? Help

Top of Chapter

Back

6 Continue creating a database checkpoint in WinRunner:

- To create a default check on a database, follow the instructions starting at step 5 on **page 334**.

- To create a custom check on a database, follow the instructions starting at step 5 on **page 339**.

For additional information on working with Data Junction, refer to the Data Junction documentation.

Books Online

Find

Find Again

Help

Top of Chapter

Back
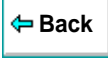
Creating a Default Check on a Database

When you create a default check on a database, you create a database checkpoint that checks the entire result set.

- The default check for a multiple-column query on a database is a case sensitive check on the entire result set by column name and row index.

- The default check for a single-column query on a database is a case sensitive check on the entire result set by row position.

If you want to check only part of the contents of a result set, edit the expected value of the contents, or count the number of rows or columns, you should create a custom check instead of a default check. For information on creating a custom check on a database, see **Creating a Custom Check on a Database** on page 337.

Books Online

Find

Find Again

Help

Top of Chapter

Back

Creating a Default Check on a Database Using ODBC or Microsoft Query

You can create a default check on a database using ODBC or Microsoft Query.

To create a default check on a database using ODBC or Microsoft Query:

1 Choose **Create > Database Checkpoint > Default Check** or click the **Default Database Checkpoint** button on the User toolbar. If you are recording in Analog mode, press the CHECK DATABASE (DEFAULT) softkey in order to avoid extraneous mouse movements. Note that you can press the CHECK DATABASE (DEFAULT) softkey in Context Sensitive mode as well.

Note: The first time you create a default database checkpoint, either Microsoft Query or the Database Checkpoint wizard opens. Each subsequent time you create a default database checkpoint, the last tool used is opened. If the Database Checkpoint wizard opens, follow the instructions in **Working with the Database Checkpoint Wizard** on page 347.

2 If Microsoft Query is installed and you are creating a new query, an instruction screen opens for creating a query.

If you do not want to see this message next time you create a default database checkpoint, clear the **Show this message next time** check box.

Click **OK** to close the instruction screen.

3 If Microsoft Query is not installed, the Database Checkpoint wizard opens to a screen where you can define the ODBC query manually. For additional information, see **Specifying an SQL Statement** on page 352.

4 Define a query, copy a query, or specify an SQL statement. For additional information, see **Choosing a Database** on page 327.

Note: If you want to be able to parameterize the SQL statement in the **db_check** statement which will be generated, then in the last wizard screen in Microsoft Query, click **View data or edit query in Microsoft Query**. Follow the instructions in **Guidelines for Parameterizing SQL Statements** on page 394.

---

5 WinRunner takes several seconds to capture the database query and restore the WinRunner window.

WinRunner captures the data specified by the query and stores it in the test's *exp* folder. WinRunner creates the *msqr*.sql* query file and stores it and the database checklist in the test's *chklist* folder. A database checkpoint is inserted in the test script as a **db_check** statement. For more information on the **db_check** function, refer to the *TSL Online Reference*.

Creating a Default Check on a Database Using Data Junction

You can create a default check on a database using Data Junction.

To create a default check on a database:

1 Choose **Create > Database Checkpoint > Default Check** or click the **Default Database Checkpoint** button on the User toolbar. If you are recording in Analog mode, press the CHECK DATABASE (DEFAULT) softkey in order to avoid extraneous mouse movements. Note that you can press the CHECK DATABASE (DEFAULT) softkey in Context Sensitive mode as well.

Note: The first time you create a default database checkpoint, either Microsoft Query or the Database Checkpoint wizard opens. Each subsequent time you create a default database checkpoint, the last client used is opened: if you used Microsoft Query, then Microsoft Query opens; if you use Data Junction, then the Database Checkpoint wizard opens. Note that the Database Checkpoint wizard must open whenever you use Data Junction to create a database checkpoint.

For information on working with the Database Checkpoint wizard, see **Working with the Database Checkpoint Wizard** on page 347.

Books Online

M Find

Find Again

? Help

Top of Chapter

⬅ Back

2 An instruction screen opens for creating a query.

If you do not want to see this message next time you create a default database checkpoint, clear the **Show this message next time** check box.

Click **OK** to close the instruction screen.

3 Create a new conversion file or use an existing one. For additional information, see **Choosing a Database** on page 327.

4 WinRunner takes several seconds to capture the database query and restore the WinRunner window.

WinRunner captures the data specified by the query and stores it in the test's *exp* folder. WinRunner creates the **.djs* conversion file and stores it in the checklist in the test's *chklist* folder. A database checkpoint is inserted in the test script as a **db_check** statement. For more information on the **db_check** function, refer to the *TSL Online Reference*.

Creating a Custom Check on a Database

When you create a custom check on a database, you create a database checkpoint in which you can specify which properties to check on a result set.

You can create a custom check on a database in order to:

- check the contents of part or the entire result set
- edit the expected results of the contents of the result set
- count the rows in the result set
- count the columns in the result set

Books Online

Find

Find Again

Help

Top of Chapter

Back

Creating a Custom Check on a Database Using ODBC or Microsoft Query

You can create a custom check on a database using ODBC or Microsoft Query.

To create a custom check on a database using ODBC or Microsoft Query:

1 Choose **Create > Database Checkpoint > Custom Check**. If you are recording in Analog mode, press the CHECK DATABASE (CUSTOM) softkey in order to avoid extraneous mouse movements. Note that you can press the CHECK DATABASE (CUSTOM) softkey in Context Sensitive mode as well.

The Database Checkpoint wizard opens.

2 Follow the instructions on working with the Database Checkpoint wizard, as described in **Working with the Database Checkpoint Wizard** on page 347.

3 If you are creating a new query, an instruction screen opens for creating a query.

If you do not want to see this message next time you create a default database checkpoint, clear the **Show this message next time** check box.

4 Define a query, copy a query, or specify an SQL statement. For additional information, see **Choosing a Database** on page 327.

Note: If you want to be able to parameterize the SQL statement in the **db_check** statement which will be generated, then in the last wizard screen in Microsoft Query, click **View data or edit query in Microsoft Query**. Follow the instructions in **Parameterizing Database Checkpoints** on page 389.

Books Online

Find

Find Again

Help

Top of Chapter

Back

5 WinRunner takes several seconds to capture the database query and restore the WinRunner window.

The **Check Database** dialog box opens.

The **Objects** pane contains "Database check" and the name of the *.sql* query file that will be included in the database checkpoint. The **Properties** pane lists the different types of checks that can be performed on the result set. A check mark indicates that the item is selected and is included in the checkpoint.

6 Select the types of checks to perform on the database. You can perform the following checks:

ColumnsCount: Counts the number of columns in the result set.

Content: Checks the content of the result set, as described in **Creating a Default Check on a Database** on page 332.

RowsCount: Counts the number of rows in the result set.

7 If you want to edit the expected value of a property, first select it. Next, either click the **Edit Expected Value** button, or double-click the value in the Expected Value column.

- For **ColumnsCount** or **RowCount** checks on a result set, the expected value is displayed in the **Expected Value** field corresponding to the property check. When you edit the expected value for these property checks, a spin box opens. Modify the number that appears in the spin box.

- For a **Content** check on a result set, <complex value> appears in the **Expected Value** field corresponding to the check, since the content of the result set is too complex to be displayed in this column. When you edit the expected value, the **Edit Check** dialog box opens. In the **Select Checks** tab, you can select which checks to perform on the result set, based on the data captured in the query. In the **Edit Expected Data** tab, you can modify the expected results of the data in the result set.

  For more information, see **Understanding the Edit Check Dialog Box** on page 359.

Books Online

Find

Find Again

Help

Top of Chapter

Back

8 Click **OK** to close the Check Database dialog box.

WinRunner captures the current property values and stores them in the test's *exp* folder. WinRunner stores the database query in the test's *chklist* folder. A database checkpoint is inserted in the test script as a **db_check** statement. For more information on the **db_check** function, refer to the *TSL Online Reference*.

Books Online

Find

Find Again

Help

Top of Chapter

Back

Creating a Custom Check on a Database Using Data Junction

You can create a custom check on a database using Data Junction.

Books Online

To create a custom check on a database using Data Junction:

M Find

1 Choose **Create > Database Checkpoint > Custom Check**. If you are recording in Analog mode, press the CHECK DATABASE (CUSTOM) softkey in order to avoid extraneous mouse movements. Note that you can press the CHECK DATABASE (CUSTOM) softkey in Context Sensitive mode as well.

Find Again

The Database Checkpoint wizard opens.

? Help

2 Follow the instructions on working with the Database Checkpoint wizard, as described in **Working with the Database Checkpoint Wizard** on page 347.

3 If you are creating a new query, an instruction screen opens.

Top of Chapter

If you do not want to see this message next time you create a default database checkpoint, clear the **Show this message next time** check box.

← Back

4 Create a new conversion file or use an existing one. For additional information, see **Choosing a Database** on page 327.

5 WinRunner takes several seconds to capture the database query and restore the WinRunner window.

The **Check Database** dialog box opens.

The **Objects** pane contains "Database check" and the name of the **.djs* conversion file that will be included in the database checkpoint. The **Properties** pane lists the different types of checks that can be performed on the result set. A check mark indicates that the item is selected and is included in the checkpoint.

6 Select the types of checks to perform on the database. You can perform the following checks:

ColumnsCount: Counts the number of columns in the result set.

Content: Checks the content of the result set, as described in **Creating a Default Check on a Database** on page 332.

RowsCount: Counts the number of rows in the result set.

7 If you want to edit the expected value of a property, first select it. Next, either click the **Edit Expected Value** button, or double-click the value in the Expected Value column.

- For **ColumnsCount** or **RowCount** checks on a result set, the expected value is displayed in the **Expected Value** field corresponding to the property check. When you edit the expected value for these property checks, a spin box opens. Modify the number that appears in the spin box.

- For a **Content** check on a result set, <complex value> appears in the **Expected Value** field corresponding to the check, since the content of the result set is too complex to be displayed in this column. When you edit the expected value, the **Edit Check** dialog box opens. In the **Select Checks** tab, you can select which checks to perform on the result set, based on the data captured in the query. In the **Edit Expected Data** tab, you can modify the expected results of the data in the result set.

For more information, see **Understanding the Edit Check Dialog Box** on page 359.

8 Click **OK** to close the Check Database dialog box.

WinRunner captures the current property values and stores them in the test's *exp* folder. WinRunner stores the database query in the test's *chklist* folder. A database checkpoint is inserted in the test script as a **db_check** statement. For more information on the **db_check** function, refer to the *TSL Online Reference*.

Books Online

Find

Find Again

Help

Top of Chapter

Back
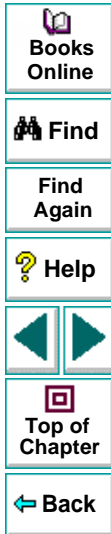
Messages in the Database Checkpoint Dialog Boxes

The following messages may appear in the Properties pane in the Expected Value or the Actual Value fields in the Check Database or the Database Checkpoint Results dialog boxes:

Message	Meaning
Complex Value	The expected or actual value of the selected property check is too complex to display in the column. This message will appear for the content checks.
Cannot Capture	The expected or actual value of the selected property could not be captured.

Note: For information on the Database Checkpoint Results dialog box, see Chapter 28, **Analyzing Test Results**.

Working with the Database Checkpoint Wizard

The wizard opens whenever you create a custom database checkpoint and whenever you work with Data Junction. You can also use an SQL statement to create a database checkpoint. When working with SQL statements, create a custom database check and choose the ODBC (Microsoft Query) option.

You can work in either ODBC/Microsoft Query mode or Data Junction mode. Depending on the last tool used, a screen opens for either ODBC (Microsoft Query) or Data Junction. You can change from one mode to another in the first wizard screen.

The Database Checkpoint wizard enables you to:

- switch between ODBC (Microsoft Query) mode and Data Junction mode
- specify an SQL statement without using Microsoft Query
- use existing queries and conversions in your database checkpoint

Books Online

Find

Find Again

Help

Top of Chapter

Back

ODBC (Microsoft Query) Screens

There are three screens in the Database Checkpoint wizard for working with ODBC (Microsoft Query). These screens enable you to:

- set general options:

  - switch to Data Junction mode

  - choose to create a new query, use an existing one, or specify an SQL statement

  - limit the number of rows in the query

  - display an instruction screen

- select an existing source query file

- specify an SQL statement

Books Online

Find

Find Again

Help

Top of Chapter

Back

Setting ODBC (Microsoft Query) Options

The following screen opens if you are creating a custom database checkpoint or working in ODBC mode.
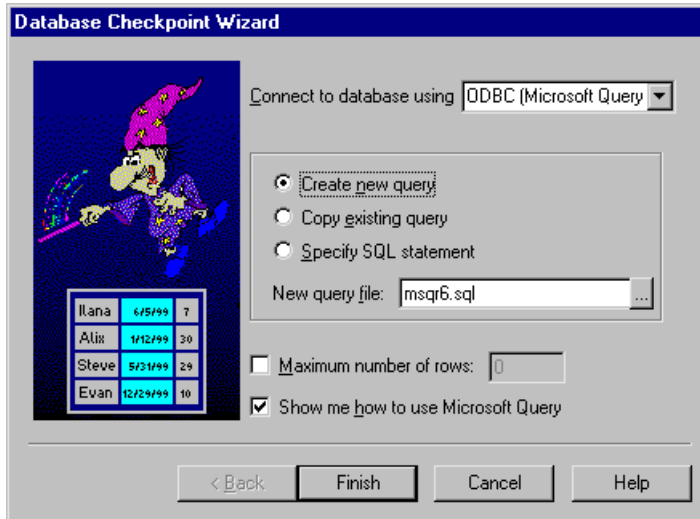
You can choose from the following options:

- **Create new query:** Opens Microsoft Query, enabling you to create a new ODBC **.sql* query file with the name specified below. Once you finish defining your query:

  - If you are creating a default database checkpoint, a **db_check** statement is inserted into your test script.

  - If you are creating a custom database checkpoint, the Check Database dialog box opens. For information on the Check Database dialog box, see **Creating a Custom Check on a Database** on page 337.

- **Copy existing query:** Opens the **Select source query file** screen in the wizard, which enables you to copy an existing ODBC query from another query file. For additional information, see **Selecting a Source Query File** on page 351.

- **Specify SQL statement:** Opens the **Specify SQL statement** screen in the wizard, which enables you to specify the connection string and an SQL statement. For additional information, see **Specifying an SQL Statement** on page 352.

- **New query file:** Displays the default name of the new **.sql* query file for this database checkpoint. You can use the browse button to browse for a different **.sql* query file.

- **Maximum number of rows:** Select this check box and enter the maximum number of database rows to check. If this check box is cleared, there is no maximum. Note that this option adds an additional parameter to your **db_check** statement. For more information, refer to the *TSL Online Reference.*

● **Show me how to use Microsoft Query:** Displays an instruction screen.

Selecting a Source Query File

The following screen opens if you chose to use an existing query file in this database checkpoint.
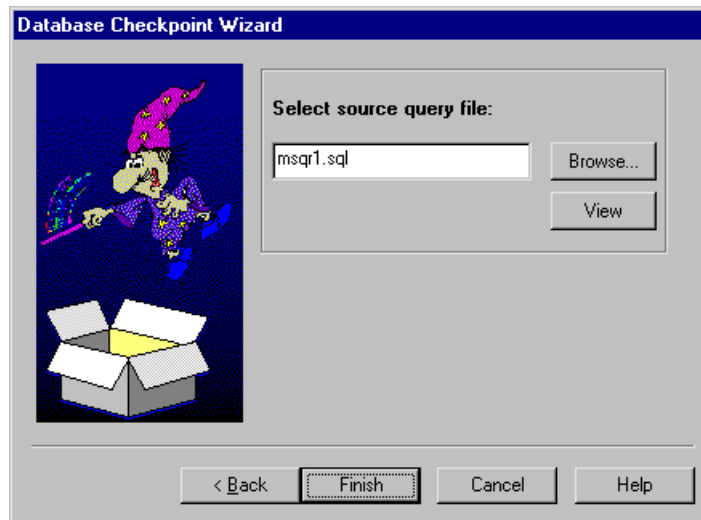
Enter the pathname of the query file or use the **Browse** button to locate it. Once a query file is selected, you can use the **View** button to open the file for viewing.
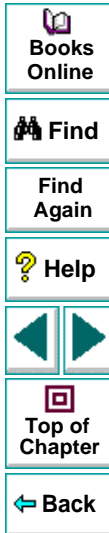
- If you are creating a default database checkpoint, a **db_check** statement is inserted into your test script.

- If you are creating a custom database checkpoint, the Check Database dialog box opens. For information on the Check Database dialog box, see **Creating a Custom Check on a Database** on page 337.

Specifying an SQL Statement

The following screen opens if you chose to specify an SQL statement to use in this database checkpoint.

In this screen you must specify the connection string and the SQL statement:

- **Connection String:** Enter the connection string, or click **Create** to open the ODBC Select Data Source dialog box, in which you can select a *.dsn* file, which inserts the connection string in the box.

- **SQL:** Enter the SQL statement.

Note: If you create an SQL statement containing parameters, an instruction screen opens. For information on parameterizing SQL statements, see **Parameterizing Database Checkpoints** on page 389.

- If you are creating a default database checkpoint, a **db_check** statement is inserted into your test script.

- If you are creating a custom database checkpoint, the Check Database dialog box opens. For information on the Check Database dialog box, see **Creating a Custom Check on a Database** on page 337.
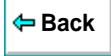
Data Junction Screens in the Database Checkpoint Wizard

There are two screens in the Database Checkpoint wizard for working with Data Junction. These screens enable you to:

- set general options:

  - switch to ODBC (Microsoft Query) mode

  - choose to create a new conversion or use an existing one

  - display an instruction screen

- specify the conversion file

Setting Data Junction Options

The following screen opens if you last worked with Data Junction or if you are creating a default database checkpoint for the first time when only Data Junction is installed:
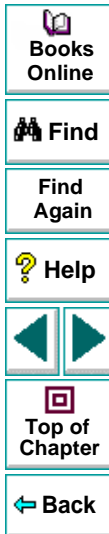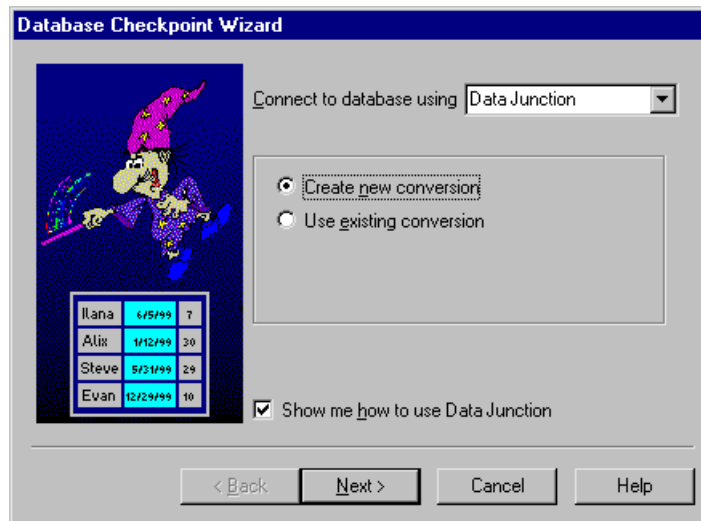
You can choose from the following options:

- **Create new conversion:** Opens Data Junction and enables you to create a new conversion file. For additional information, see **Creating a Conversion File in Data Junction** on page 330. Once you have created a conversion file, the Database Checkpoint wizard screen reopens to enable you to specify this file. For additional information, see **Selecting a Data Junction Conversion File** on page 357

- **Use existing conversion:** Opens the **Select conversion file** screen in the wizard, which enables you to specify an existing conversion file. For additional information, see **Selecting a Data Junction Conversion File** on page 357.

- **Show me how to use Data Junction** (available only when **Create new conversion** is selected): Displays instructions for working with Data Junction.

Selecting a Data Junction Conversion File

The following wizard screen opens when you are working with Data Junction.

Enter the pathname of the conversion file or use the **Browse** button to locate it.
Once a conversion file is selected, you can use the **View** button to open the file
for viewing.

You can also choose from the following options:

- **Copy conversion to test folder:** Copies the specified conversion file to the test folder

- **Maximum number of rows:** Select this check box and enter the maximum number of database rows to check. If this check box is cleared, there is no maximum.

When you are done:

- If you are creating a default database checkpoint, a **db_check** statement is inserted into your test script.

- If you are creating a custom database checkpoint, the Check Database dialog box opens. For information on the Check Database dialog box, see **Creating a Custom Check on a Database** on page 337.

Books Online

Find

Find Again

Help

Top of Chapter

Back

Understanding the Edit Check Dialog Box

The **Edit Check** dialog box enables you to specify which cells to check, and which verification method and verification type to use. You can also edit the expected data for the database cells included in the check.

Books Online

Find

Find Again

Help

Top of Chapter

Back

Edit Check

Select Checks | Edit Expected Data

1. Select the rows, columns or cells to check:

Auto Size

	Flight	From	Departure	To	Arrival	Ariline	Price	col_7
1	9039	LAX	05:43 PM	POR	07:24 PM	SW	$130.40	
2	8678	LAX	08:07 AM	POR	09:48 AM	SW	$150.00	
3	7302	LAX	12:55 PM	POR	02:36 PM	SW	$154.80	
4	6802	LAX	04:31 PM	POR	06:12 PM	SW	$141.20	
5	6090	LAX	09:19 AM	POR	11:00 AM	SW	$149.20	
6	4397	LAX	03:19 PM	POR	05:00 PM	SW	$138.40	

2. Set verification type: Case Sensitive

3. Add | Cell [column 'Flight', row 1] - Case Sensitive check

List of checks: | Entire Table - Case Sensitive check

Delete

Verification methods:

Column
- ⦿ Name
- ○ Index

☐ Verify column headers

Row
- ○ Key
- ⦿ Index

Select key columns:
Flight
From
Departure
To

OK | Cancel | Help

Ready

Books Online

Find

Find Again

Help

Top of Chapter

Back

In the **Selected Checks** tab, you can specify the information that is saved in the database checklist:

- which database cells to check

- the verification method

- the verification type

Note that if you are creating a check on a single-column result set, the contents of the Select Checks tab of the Edit Check dialog box differ from what is shown above. For additional information, see **Specifying the Verification Method for a Single-Column Result Set** on page 365.

Specifying which Cells to Check

The **List of Checks** pane displays all the checks that will be performed, including the verification type. When the Edit Check dialog box is opened for the first time for a checkpoint, the default check is displayed:

- The default check for a multiple-column result set is a case sensitive check on the entire result set by column name and row index.

- The default check for a single-column result set is a case sensitive check on the entire result set by row position.

Note: If your result set contains multiple columns with the same name, WinRunner disregards the duplicate columns and does not perform checks on them. Therefore, you should create a custom check on the database and select the column index option.

If you do not wish to accept the default settings, you must delete the default check before you specify the checks to perform. Select the "Entire Table - Case Sensitive check" entry in the **List of Checks** pane and click the **Delete** button. Alternatively, double-click this entry in the **List of Checks** pane. A WinRunner message prompts you to delete the highlighted check. Click **Yes**.

Next, specify the checks to perform. You can choose different verification types for different selections of cells. Therefore, specify the verification type before selecting cells. For more information, see **Specifying the Verification Type** on page 367.

Highlight the cells on which you want to perform the content check. Next, click the **Add** button to add a check for these cells. Alternatively, you can:

- double-click a cell to check it

- double-click a row header to check all the cells in a row

- double-click a column header to check all the cells in a column

- double-click the top-left corner to check the entire result set

A description of the cells to be checked appears in the **List of Checks** pane.

Books Online

Find

Find Again

Help

Top of Chapter

Back

Specifying the Verification Method

You can select the verification method to control how WinRunner identifies columns or rows within a result set. The verification method applies to the entire result set. Specifying the verification method is different for multiple-column and single-column result sets.

Specifying the Verification Method for a Multiple-Column Result Set

Column

- **Name:** (default setting): WinRunner looks for the selection according to the column names. A shift in the position of the columns within the result set does not result in a mismatch.

- **Index:** WinRunner looks for the selection according to the index, or position, of the columns. A shift in the position of the columns within the result set results in a mismatch. Select this option if your result set contains multiple columns with the same name. For additional information, see the note on **page 362**. Choosing this option enables the **Verify column headers** check box, which enables you to check column headers as well as cells.

Row

- **Key:** WinRunner looks for the rows in the selection according to the key(s) specified in the **Select key columns** list box, which lists the names of all columns in the result set. A shift in the position of any of the rows does not result in a mismatch. If the key selection does not identify a unique row, only the first matching row will be checked.

- **Index:** (default setting): WinRunner looks for the selection according to the index, or position, of the rows. A shift in the position of any of the rows results in a mismatch.

Specifying the Verification Method for a Single-Column Result Set

The Verification Method box in the Select Checks tab for a single-column result set is different from that for a multiple-column result set. The default check for a single-column result set is a case sensitive check on the entire result set by row position.

- **By position:** WinRunner checks the selection according to the location of the items within the column.

- **By content:** WinRunner checks the selection according to the content of the items, ignoring their location in the column.

Edit Check

Select Checks	Edit Expected Data

1. Select the rows, columns or cells to check: Auto Size

	Departure
4	San Francisco
5	Seattle
6	Denver
7	Seattle
8	Denver
9	Los Angeles

2. Set verification type: Case Sensitive

3. Add   Cell [column 'Departure', row 1] - Case Sensitive check

List of checks: Entire Table - Case Sensitive check

Delete

Verification methods: ⦿ By position
                         ○ By content

OK   Cancel   Help

Ready

Books Online

Find

Find Again

Help

Top of Chapter

Back
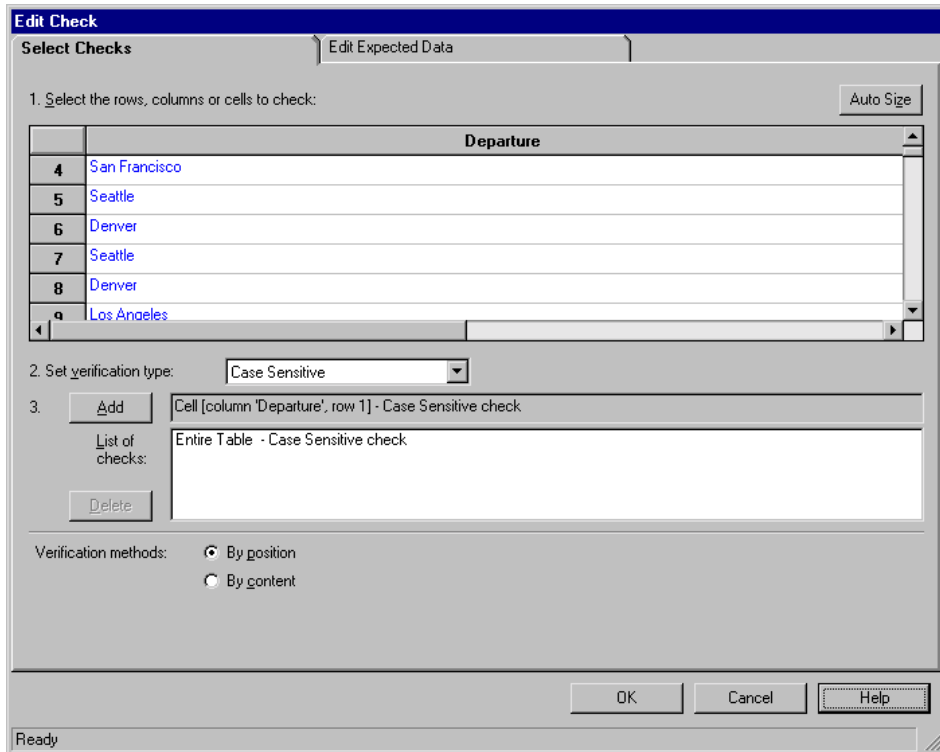
Specifying the Verification Type

WinRunner can verify the contents of a result set in several different ways. You can choose different verification types for different selections of cells.

- **Case Sensitive** (the default): WinRunner compares the text content of the selection. 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. 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.

- **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 database data is compared against the range that you defined and not against the expected results.

- **Case Sensitive Ignore Spaces:** WinRunner checks the data in the field according to case and content, ignoring differences in spaces. WinRunner reports any differences in case or content as a mismatch.

- **Case Insensitive Ignore Spaces:** WinRunner checks the content in the cell according to content, ignoring differences in case and spaces. WinRunner reports only differences in content as a mismatch.

Books Online

Find

Find Again

Help

Top of Chapter

Back

Click **OK** to save your changes to both tabs of the Edit Check dialog box. The dialog box closes and the Check Database dialog box is restored.

Editing the Expected Data

To edit the expected data in the result set, click the **Edit Expected Data** tab. If you previously saved changes in the Select Checks tab, you can click **Reload Table** to reload the selections from the checklist. A WinRunner message prompts you to reload the saved data. Click **Yes**.

Note that if you previously saved changes to the Select Checks tab, and then reopened the Edit Check dialog box, the table appears color coded in the Edit Expected Data tab. The cells included in the check appear in blue on a white background. The cells excluded from the check appear in green on a yellow background.

Edit Check

Select Checks | Edit Expected Data

	Flight	From	Departure	To	Arrival	Ariline	Price	col_7
1	8961	LAX	10:31 AM	POR	12:12 PM	UA	$121.60	
2	8564	LAX	02:07 PM	POR	03:48 PM	UA	$121.20	
3	7845	LAX	08:07 AM	POR	09:48 AM	UA	$147.60	
4	7826	LAX	09:19 AM	POR	11:00 AM	UA	$124.80	
5	7173	LAX	04:31 PM	POR	06:12 PM	UA	$135.20	
6	7148	LAX	03:19 PM	POR	05:00 PM	UA	$130.40	
7	7072	LAX	12:55 PM	POR	02:36 PM	UA	$158.00	
8	6791	LAX	06:55 PM	POR	08:36 PM	UA	$122.80	
9	4302	LAX	03:12 PM	POR	05:12 PM	TWA	$162.40	
10	4298	LAX	12:48 PM	POR	02:48 PM	TWA	$168.50	
11	4294	LAX	10:24 AM	POR	12:24 PM	TWA	$162.30	
12	4290	LAX	08:00 AM	POR	10:00 AM	TWA	$160.40	
13	2730	LAX	05:43 PM	POR	07:24 PM	UA	$130.80	
14	1365	LAX	11:43 AM	POR	01:24 PM	UA	$124.40	

OK | Cancel | Help

Ready

Books Online

Find

Find Again

Help

Top of Chapter

Back

To edit the expected value of data in a cell, double-click inside the cell. A cursor appears in the cell. Change the contents of the cell, as desired. Click **OK** to save your changes to both tabs of the Edit Check dialog box. The dialog box closes and the Check Database dialog box is restored.
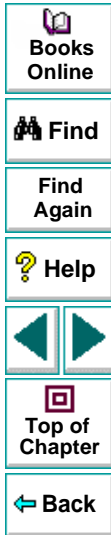
Modifying a Database Checkpoint

You can make changes to an existing database checkpoint.

You can:

- make a checklist available to other users by saving it in a shared folder

- change which database properties to check in an existing checklist

- modify a query in an existing checklist

Note: In addition to modifying database checklists, you can also modify the expected results of database checkpoints. For more information, see **Modifying the Expected Results of a Database Checkpoint** on page 386.

Books Online

Find

Find Again

Help

Top of Chapter

Back

Saving a Database Checklist in a Shared Folder

By default, checklists for database checkpoints are stored in the folder of the current test. You can specify that a checklist be placed in a shared folder to enable wider access, so that you can use a checklist in multiple tests.

The default folder in which WinRunner stores your shared checklists is *WinRunner installation folder/chklist*. To choose a different folder, you can use the **Shared Checklists** box in the Folders tab of the General Options dialog box. For more information, see Chapter 36, **Setting Global Testing Options**.

To save a database checklist in a shared folder:

1 Choose **Create** > **Edit Database Checklist**.

The Open Checklist dialog box opens.
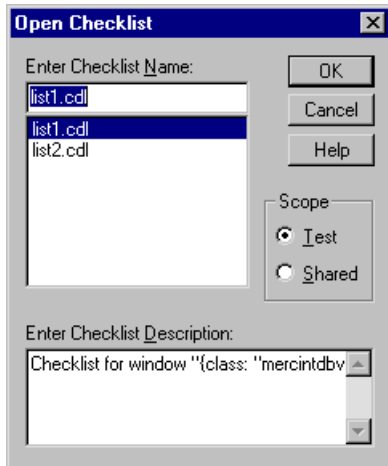
2 Select a database checklist and click **OK**. Note that database checklists have the .cdl extension, while GUI checklists have the .ckl extension. For information on GUI checklists, see **Modifying GUI Checklists** on page 215.

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

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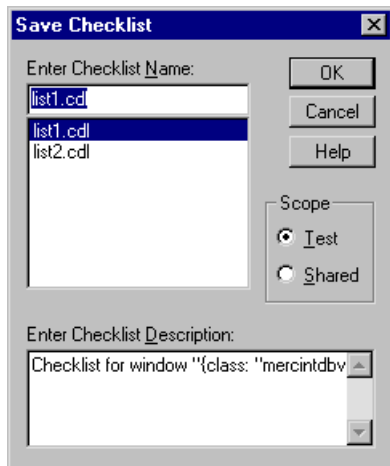
The Save Checklist dialog box opens.

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

5 Click **OK** to close the Edit Database Checklist dialog box.

Editing Database Checklists

You can edit an existing database checklist. Note that a database checklist includes only a reference to the *msqr*.sql* query file or the **.djs* conversion file of the database and the properties to be checked. It does not include the expected results for the values of those properties.

You may want to edit a database checklist to change which properties in a database to check.

Books Online

Find

Find Again

? Help

Top of Chapter

⇐ Back

To edit an existing database checklist:

 1 Choose **Create** > **Edit Database Checklist**. The Open Checklist dialog box opens.

 2 A list of checklists for the current test is displayed. If you want to see checklists in a shared folder, click **Shared**.

For more information on sharing database checklists, see **Saving a Database Checklist in a Shared Folder** on page 372.

Lists the available checklists.

Displays checklists created for the current test.

Displays checklists created in a shared folder.

Describes the selected checklist.

Books Online

Find

Find Again

Help

Top of Chapter

Back
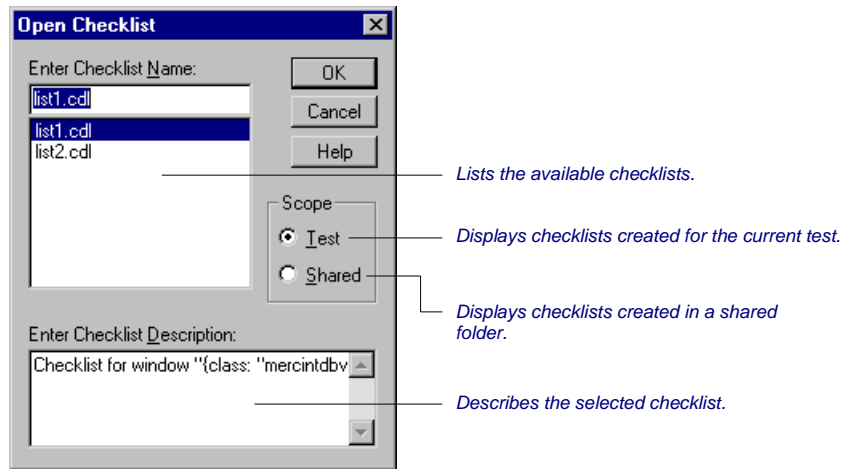
 3 Select a database checklist.

4 Click **OK**.

The Open Checklist dialog box closes. The Edit Database Checklist dialog box opens and displays the selected checklist.

The **Objects** pane contains "Database check" and the name of the **.sql* query file or **.djs* conversion file that will be included in the database checkpoint. The **Properties** pane lists the different types of checks that can be performed on databases. A check mark indicates that the item is selected and is included in the checkpoint.

Name of *.sql query file or *.djs conversion file
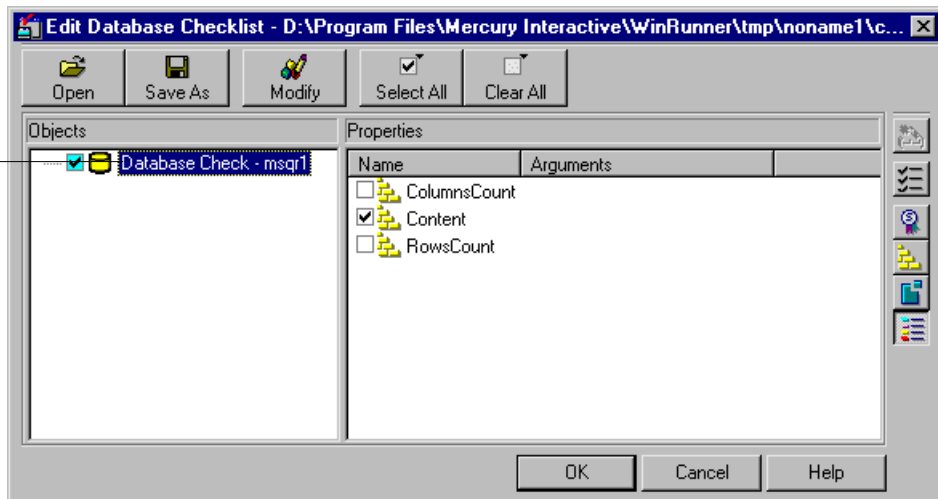
You can use the **Modify** button to modify the database checkpoint, as described in **Modifying a Query in an Existing Database Checklist** on page 379.

In the **Properties** pane, you can edit your database checklist to include or exclude the following types of checks:

ColumnsCount: Counts the number of columns in the result set.

Content: Checks the content of the result set, as described in **Creating a Default Check on a Database** on page 332.

RowsCount: Counts the number of rows in the result set.

5 Save the checklist in one of the following ways:

● To save the checklist under its existing name, click **OK** to close the Edit Database Checklist dialog box. A WinRunner message prompts you to overwrite the existing checklist. Click **OK**.

● To save the checklist under a different name, 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 Database Checklist dialog box.

A new database checkpoint statement is *not* inserted in your test script.

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 Run Modes** on page 677.

Modifying a Query in an Existing Database Checklist

You can modify a query in an existing database checklist from the Edit Database Checklist dialog box. You may want to do this if, for example, you move the database to a new location on the network. You must use the same tool to modify the query that you used to create it.

Modifying a Query Created with ODBC/Microsoft Query

You can modify a query created with ODBC/Microsoft Query from the Edit Database Checklist dialog box.

To modify a database checkpoint created with ODBC/Microsoft Query:

 1 Choose **Create > Edit Database Checklist**. The Open Checklist dialog box opens.

 2 A list of checklists for the current test is displayed. If you want to see checklists in a shared folder, click **Shared**.

For more information on sharing database checklists, see **Saving a Database Checklist in a Shared Folder** on page 372.

Open Checklist

Enter Checklist Name:

list1.cdl

OK

Cancel

Help

list1.cdl
list2.cdl

Lists the available checklists.

Scope

◉ Test

○ Shared

Displays checklists created for the current test.

Displays checklists created in a shared folder.

Enter Checklist Description:

Checklist for window ''{class: ''mercintdbv
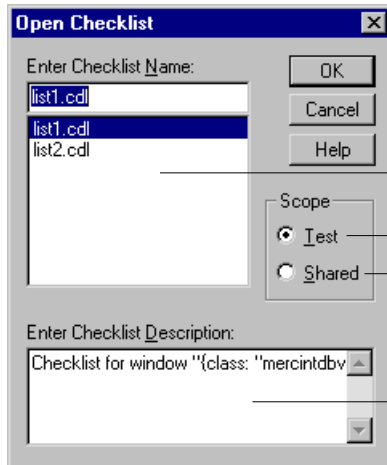
Describes the selected checklist.

3 Select a database checklist.

4 Click **OK**.

The Open Checklist dialog box closes. The Edit Database Checklist dialog box opens and displays the selected checklist.

The **Objects** pane contains "Database check" and the name of the **.sql* query file that will be included in the database checkpoint. The **Properties** pane lists the different types of checks that can be performed on databases. A check mark indicates that the item is selected and is included in the checkpoint. To modify the properties to check, see **Editing Database Checklists** on page 375.
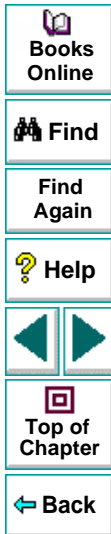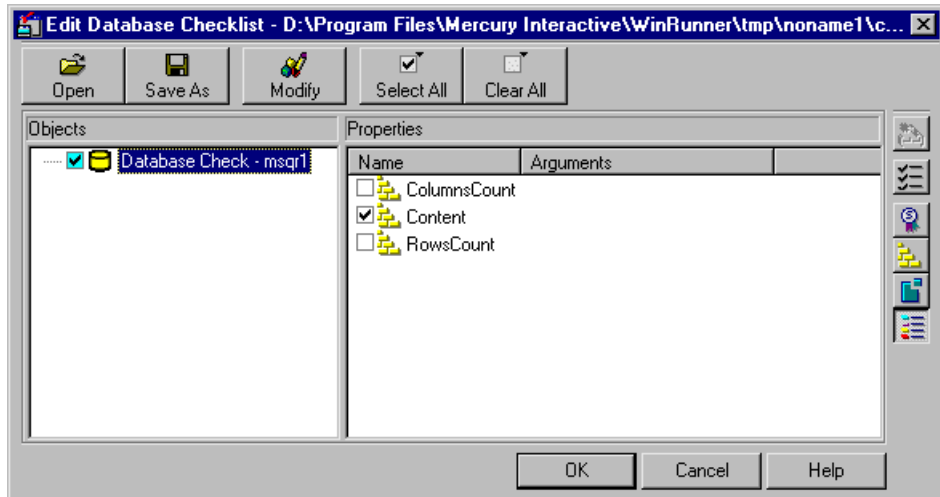
5 In the **Objects** column, highlight the name of the query file or the conversion file, and click **Modify**.

The Modify ODBC Query dialog box opens.
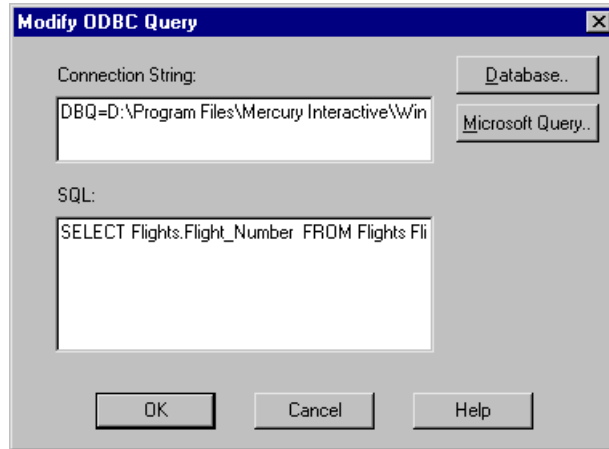
6 Modify the ODBC query by changing the connection string and/or the SQL statement. You can click **Database** to open the ODBC Select Data Source dialog box, in which you can select a **.dsn* file, which inserts the connection string in the box. You can click **Microsoft Query** to open Microsoft Query.

7 Click **OK** to return to the Edit Checklist dialog box.

8 Click **OK** to close the Edit Checklist dialog box.

Note: You must run all tests that use this checklist in Update mode before you run them in Verify mode.

Modifying a Query Created with Data Junction

You can modify a Data Junction conversion file used in a database checkpoint directly in Data Junction. To see the pathname of the conversion file, follow the instructions below.

To see the pathname of a Data Junction conversion file in a database checkpoint:

1 Choose **Create > Edit Database Checklist**. The Open Checklist dialog box opens.

2 A list of checklists for the current test is displayed. If you want to see checklists in a shared folder, click **Shared**.

For more information on sharing database checklists, see **Saving a Database Checklist in a Shared Folder** on page 372.

Lists the available checklists.

Displays checklists created for the current test.

Displays checklists created in a shared folder.

Describes the selected checklist.

Books Online

Find

Find Again

Help

Top of Chapter

Back

3 Select a database checklist.

4 Click **OK**.

The Open Checklist dialog box closes. The Edit Database Checklist dialog box opens and displays the selected checklist.

The **Objects** pane contains "Database check" and the name of the *.djs* conversion file that will be included in the database checkpoint. The **Properties** pane lists the different types of checks that can be performed on databases. A check mark indicates that the item is selected and is included in the checkpoint. To modify the properties to check, see **Editing Database Checklists** on page 375.

Modify

5 In the **Objects** column, highlight the name of the conversion file, and click **Modify**.

Displays a message to use Data Junction to modify the conversion file and the pathname of the conversion file.

6 Click **OK** to close the message and return to the Edit Checklist dialog box.

7 Click **OK** to close the Edit Checklist dialog box.

8 Modify the conversion file directly in Data Junction.

Note: You must run all tests that use this checklist in Update mode before you run them in Verify mode.

Books Online

M Find

Find Again

? Help

Top of Chapter

Back

Modifying the Expected Results of a Database Checkpoint

You can modify the expected results of an existing database checkpoint by changing the expected value of a property check within the checkpoint. You can make this change before or after you run your test script.

To modify the expected results for an existing database checkpoint:

1 Choose **Tools > Test Results** or click **Test Results**.

The WinRunner Test Results window opens.
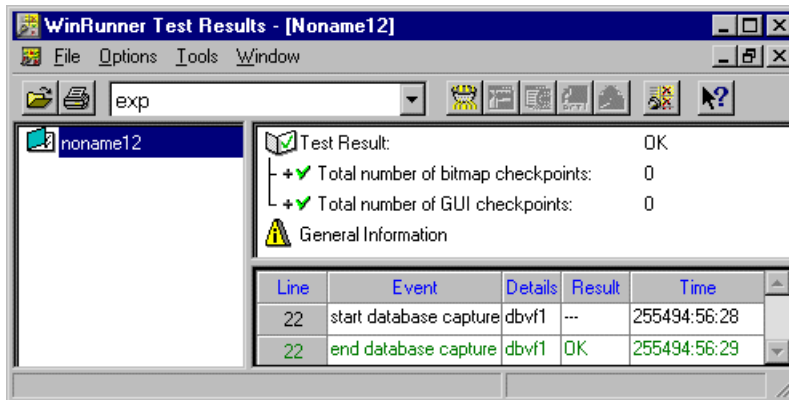
2 In the **Results** box, choose your expected results directory (by default, "exp").

3 In the test log, locate the database checkpoint by looking for entries that list "end database capture" in the **Event** column. Note that the line number in the test script appears in the **Line** column of the test log.

Note: You can use the **Show TSL** button to open the test script to the highlighted line number.

4 Double-click the desired "end database capture" entry, or click this entry and click **Display**.

The **Database Checkpoint Results** dialog box opens.

*Name of *.sql query file or *.djs conversion file*
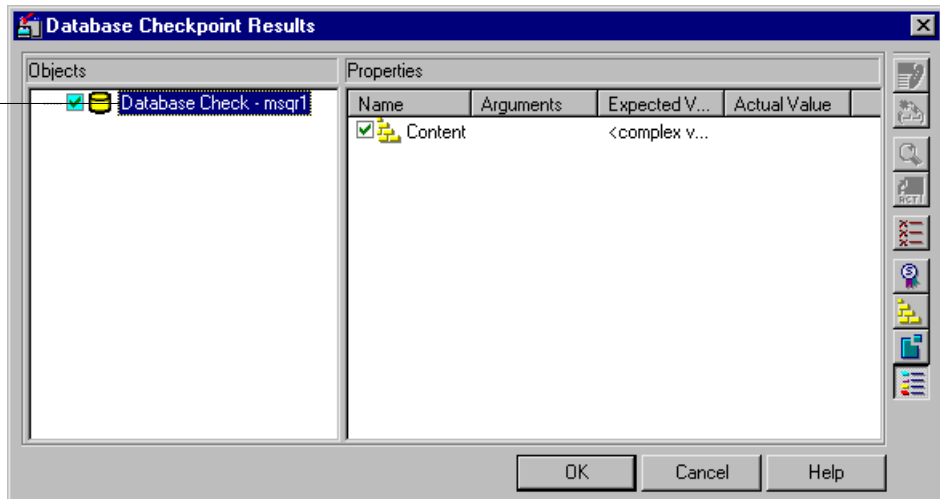
5 Select the property check whose expected results you want to modify. Click the **Edit expected value** button. In the **Expected Value** column, modify the value, as desired. Click **OK** to close the dialog box.

Note: You can also modify the expected value of a property check while creating a database checkpoint. For more information, see **Creating a Custom Check on a Database** on page 337.

Note: You can also update the expected value of a database checkpoint to the actual value after a test run. For more information, see **Updating the Expected Results of a Checkpoint** on page 736.

Parameterizing Database Checkpoints

When you create a database checkpoint using ODBC (Microsoft Query), you can add parameters to an SQL statement to parameterize your checkpoint. This is useful if you want to create a database checkpoint with a query in which the SQL statement defining your query changes. For example, suppose you are working with the sample Flight application, and you want to select all the records of flights departing from San Francisco when you create the query. You might also want to use an identical query to check all the flights departing from Denver. Instead of creating a new query or rewriting the SQL statement in the existing query to reflect this change, you can parameterize the SQL statement so that you can use a parameter for the departure value. You can replace the parameter with either value, "San Francisco," or "Denver."

When you create a database checkpoint, you insert a **db_check** statement into your test script. When you parameterize the SQL statement in your checkpoint, the **db_check** function has a fourth, optional parameter, which is an array argument. You insert corresponding array statements in your test script, before the **db_check** statement.

The syntax of the SQL statement does not assign a variable name to each parameter. In an SQL statement generated in Microsoft Query, the bracket symbols are replaced by a question mark symbol (**?**). The SQL interpreter will replace the question mark symbol with the content of the parameter. If you are working with more than one parameter in an SQL statement, you specify an ordered list of parameters, and the interpreter replaces the symbols with the parameters.

Books Online

Find

Find Again

Help

To insert a parameterized SQL statement into a db_check statement:

1 Create the parameterized SQL statement using one of the following methods.

- In Microsoft Query, once you have defined your query, add criteria whose values are a set of square brackets (**[ ]**). When you are done, click **File > Exit and return to WinRunner**. It may take several seconds to return to WinRunner.

Top of Chapter

Back

- If you are working with ODBC, enter a parameterized SQL statement, with a question mark symbol (**?**) in place of each parameter, in the Database Checkpoint wizard. For additional information, see **Specifying an SQL Statement** on page 352.

2 Finish creating the database checkpoint.

- If you are creating a *default* database checkpoint, WinRunner captures the database query.

- If you are creating a *custom* database checkpoint, the Check Database dialog box opens. You can select which checks to perform on the database. For additional information, see **Creating a Custom Check on a Database Using ODBC or Microsoft Query** on page 338. Once you close the Check Database dialog box, WinRunner captures the database query.

Note: If you are creating a *custom* database checkpoint, then when you try to close the Check Database dialog box, you are prompted with the following message: "The expected value of one or more selected checks is not valid. Continuing might cause these checks to fail. Do you wish to modify your selection?" Click **No**. (This message appears because <Cannot Capture> appears under the Expected Value column in the dialog box. In fact, WinRunner only finishes capturing the database query once you specify a value and run your test in Update mode.) For additional information on messages in the Check Database dialog box, see 7**Messages in the Database Checkpoint Dialog Boxes** on page 346.

3 A message box prompts you with instructions, which are also described below. Click **OK** to close the message box.

The WinRunner window is restored and a **db_check** statement similar to the following is inserted into your test script.

db_check("list1.cdl", "dbvf1", NO_LIMIT, dbvf1_params);

4 Create arrays to provide values for the variables in the SQL statement, and insert these statements above the **db_check** statement. For example, you could insert the following lines in your test script:

dbvf1_params[1] = 1100;
dbvf1_params[2] = Monday;

These arrays replace question marks (**?**) in the SQL statement above with the new values. Follow the guidelines below for adding arrays in TSL to parameterize your SQL statements.

5 Run your test in Update mode to update the SQL statement with these values.

After you have completed this procedure, you can run your test in Verify mode with the SQL statement. To change the parameters in the SQL statement, you modify the arrays in TSL.

Books Online

Find

Find Again

Help

Top of Chapter

Back

Understanding SQL Statements

You can click the SQL button in Microsoft Query to view the SQL statement which will be generated, based on the criteria you add to your query. An SQL statement contains two mandatory and one optional parameter:

- **SELECT** defines the columns to include in the query
- **FROM** specifies the path of the database
- **WHERE** (optional) specifies the conditions, or filters to use in the query

For example, the following SQL statement is based on a query on the database in the sample Flight Reservation application.

SELECT Flights.Departure, Flights.Flight_Number, Flights.Day_Of_Week
FROM Flights Flights
WHERE (Flights.Departure=?)

When you are finished parameterizing the SQL statement, you exit Microsoft Query and return to WinRunner.

A **db_check** statement containing four parameters is inserted into your test script.

db_check("list1.cdl", "dbvf1", NO_LIMIT, dbvf1_params);

Next you insert an array statement above the generated **db_check** statement which provides values for the variables in the SQL statement.

Before you can run your test in Verify mode, you must run it in Update mode.

Guidelines for Parameterizing SQL Statements

Follow the guidelines below when parameterizing SQL statements in **db_check** statements:

● If the column is numeric, the parameter value can be either a text string or a number.

● If the column is textual and the parameter value is textual, it can be a simple text string.

● If the column is textual and the parameter value is a number, it should be enclosed in simple quotes (' '), e.g. "'100'". Otherwise the user will receive a syntax error.

● Special syntax is required for dates, times, and time stamps, as shown below:

Date	{d '1999-07-11'}
Time	{t '19:59:27'}
Time Stamp	{ts '1999-07-11 19:59:27'}

Note: The date and time format may change from one ODBC driver to another.

Using TSL Functions to Work with a Database

WinRunner provides several TSL functions (**db_**) that enable you to work with databases.

You can use the Function Generator to insert the database 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 20, **Generating Functions**. For more information about these functions, refer to the *TSL Online Reference*.

Checking Data in a Database

You use the **db_check** function to create a database checkpoint with ODBC (Microsoft Query) and Data Junction. For information on this function, see **Creating a Default Check on a Database** on page 332 and **Creating a Custom Check on a Database** on page 337. For information on parameterizing **db_check** statements, see **Parameterizing Database Checkpoints** on page 389.

TSL Functions for Working with ODBC (Microsoft Query)

When you work with ODBC (Microsoft Query), you must perform the following steps in the following order:

1 Connect to the database.

2 Execute a query and create a result set based an SQL statement. (This step is optional. You must perform this step only if you do not create and execute a query using Microsoft Query.)

3 Retrieve information from the database.

4 Disconnect from the database.

The TSL functions for performing these steps are described below:

5 Connecting to a Database

The **db_connect** function creates a new database session and establishes a connection to an ODBC database. This function has the following syntax:

db_connect (*session_name, connection_string* **);**

The *session_name* is the logical name of the database session. The *connection_string* is the connection parameters to the ODBC database.

Books Online

Find

Find Again

Help

Top of Chapter

Back

6 Executing a Query and Creating a Result Set Based on an SQL Statement

The **db_execute_query** function executes the query based on the SQL statement and creates a record set. This function has the following syntax:

db_execute_query (*session_name***,** *SQL***,** *record_number* **);**

The *session_name* is the logical name of the database session. The *SQL* is the SQL statement. The *record_number* is an out parameter returning the number of records in the result set.

7 Retrieving Information from the Database

Returning the Value of a Single Field in the Database

The **db_get_field_value** function returns the value of a single field in the database. This function has the following syntax:

db_get_field_value (*session_name***,** *row_index***,** *column* **);**

The *session_name* is the logical name of the database session. The *row_index* is the numeric index of the row. (The first row is always numbered "0".) The *column* is the name of the field in the column or the numeric index of the column within the database. (The first row is always numbered "0".)

Returning the Content and Number of Column Headers

The **db_get_headers** function returns the number of column headers in a query and the content of the column headers, concatenated and delimited by tabs. This function has the following syntax:

db_get_headers (*session_name***,** *header_count***,** *header_content* **);**

The *session_name* is the logical name of the database session. The *header_count* is the number of column headers in the query. The *header_content* is the column headers, concatenated and delimited by tabs.

Returning the Row Content

The **db_get_row** function returns the content of the row, concatenated and delimited by tabs. This function has the following syntax:

db_get_row (*session_name***,** *row_index***,** *row_content* **);**

The *session_name* is the logical name of the database session. The *row_index* is the numeric index of the row. (The first row is always numbered "0".) The *row_content* is the row content as a concatenation of the fields values, delimited by tabs.

Writing the Record Set into a Text File

The **db_write_records** function writes the record set into a text file delimited by tabs. This function has the following syntax:

db_write_records (*session_name***,** *output_file* **[,** *headers* **[,** *record_limit* **]]);**

The *session_name* is the logical name of the database session. The *output_file* is the name of the text file in which the record set is written. The *headers* are an optional Boolean parameter that will include or exclude the column headers from the record set written into the text file. The *record_limit* is the maximum number of records in the record set to be written into the text file. A value of NO_LIMIT (the default value) indicates there is no maximum limit to the number of records in the record set.

Returning the Last Error Message of the Last Operation

The **db_get_last_error** function returns the last error message of the last ODBC or Data Junction operation. This function has the following syntax:

db_get_last_error (*session_name***,** *error* **);**

The *session_name* is the logical name of the database session. The *error* is the error message.

8 Disconnecting from a Database

The **db_disconnect** function disconnects WinRunner from the database and ends the database session. This function has the following syntax:

db_disconnect (*session_name* **);**

The *session_name* is the logical name of the database session.

Books Online

Find

Find Again

Help

Top of Chapter

Back

TSL Functions for Working with Data Junction

You can use the following two functions when working with Data Junction.

Running a Data Junction Export File

The **db_dj_convert** function runs a Data Junction export file (.djs file). This function has the following syntax:

db_dj_convert (*djs_file* **[,** *output_file* **[,** *headers* **[,** *record_limit* **]]]);**

The *djs_file* is the Data Junction export file. The *output_file* is an optional parameter to override the name of the target file. The *headers* are an optional Boolean parameter that will include or exclude the column headers from the Data Junction export file. The *record_limit* is the maximum number of records that will be converted.

Returning the Last Error Message of the Last Operation

The **db_get_last_error** function returns the last error message of the last ODBC or Data Junction operation. This function has the following syntax:

db_get_last_error (*session_name*, *error* **);**

The *session_name* is ignored when working with Data Junction. The *error* is the error message.

Books Online

Find

Find Again

Help

Top of Chapter

Back

Creating Tests
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 folder (*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.

Books Online

Find

Find Again

Help

Top of Chapter

Back

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.
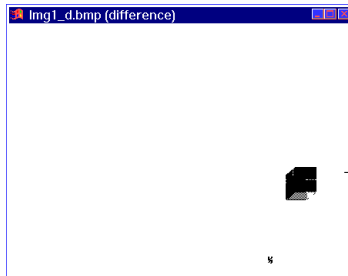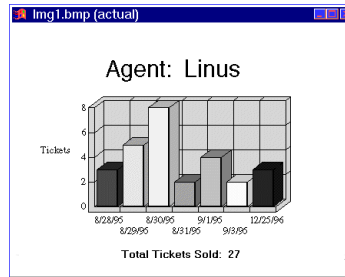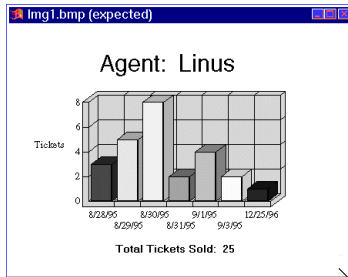
In the expected graph, captured when the test was created, 25 tickets were sold.

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 choosing Create > Bitmap Checkpoint. 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 OF WINDOW softkey or the CHECK BITMAP OF SCREEN 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 for XRunner users: 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**.

Note about data-driven testing: In order to use bitmap checkpoints in data-driven tests, you must parameterize the statements in your test script that contain them. For information on using bitmap checkpoints in data-driven tests, see **Data-Driven Checkpoints and Bitmap Synchronization Points** on page 493.

Books Online

M Find

Find Again

? Help

Top of Chapter

Back

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 **Create > Bitmap Checkpoint > For Object/Window**. 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 **Create** > **Bitmap Checkpoint > For Object/Window** or click the **Bitmap Checkpoint for Object/Window** button on the User toolbar. Alternatively, if you are recording in Analog mode, press the CHECK BITMAP OF OBJECT/WINDOW softkey.

The WinRunner window is minimized, 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**);**
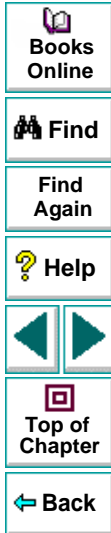
For an object bitmap, the syntax is:

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_msec*, *timeout_msec* and *min_diff* testing options. For more information on these testing options and how to modify them, see Chapter 37, **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 General Options dialog box. For more information, see Chapter 36, **Setting Global Testing Options**.

Books Online

Find

Find Again

Help

Top of Chapter

Back

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).

To capture an area of the screen as a bitmap:

1 Choose **Create > Bitmap Checkpoint > For Screen Area** or click the **Bitmap Checkpoint for Screen Area** button. Alternatively, if you are recording in Analog mode, press the CHECK BITMAP OF SCREEN AREA softkey.

   The WinRunner window is minimized, the mouse pointer becomes a crosshairs pointer, and a help window opens.

2 Mark the area to be captured: press the left mouse button and drag the mouse pointer until a rectangle encloses the area; then release the mouse button.

3 Press the right mouse button to complete the operation. WinRunner captures the area and generates a **win_check_bitmap** statement in your script.

Books Online

Find

Find Again

? Help

Top of Chapter

Back

Note: Execution of the **win_check_bitmap** function is affected by the current settings specified for the *delay_msec*, *timeout_msec* and *min_diff* test options. For more information on these testing options and how to modify them, see Chapter 37, **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 General Options dialog box. For more information, see Chapter 36, **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*.

Books Online

Find

Find Again

Help

Top of Chapter

Back

WinRunner enables you to read and check text in a GUI object or in any area of your application.

This chapter describes:

- **Reading Text**
- **Searching for Text**
- **Comparing Text**
- **Teaching Fonts to WinRunner**

Books Online

Find

Find Again

Help

Top of Chapter

Back
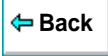
About Checking Text

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 a 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 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 428.

Books Online

M Find

Find Again

? Help

Top of Chapter

← Back

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 a **Create** > **Get Text** command, 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 **Create** > **Get Text** > **From Object/Window** and click an object with the mouse pointer. To read the text in an area of an object or window, you choose **Create** > **Get Text** > **From Screen 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 428.

Books Online

Find

Find Again

Help

Top of Chapter

Back

Reading All the Text in a Window or an 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 an object:

 1 Choose **Create > Get Text > From Object/Window** or click the **Get Text from Object/Window** button on the User toolbar. Alternatively, if you are recording in Analog mode, press the GET TEXT FROM OBJECT/WINDOW softkey.

WinRunner is minimized, 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 you choose **Create > Get Text > From 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);

Books Online

Find

Find Again

Help

Top of Chapter

Back

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.**

Note: When the WebTest add-in is loaded and a Web object is selected, WinRunner generates a **web_frame_get_text** or **web_obj_get_text** statement in your test script. For more information, refer to the *WebTest User's Guide* and the *TSL Online Reference*.

Books Online

M Find

Find Again

? Help

Top of Chapter

⇐ Back

Reading the Text from an Area of an Object or a 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 an object:

1 Choose **Create > Get Text > From Screen Area** or click the **Get Text from Screen Area** button on the User toolbar. Alternatively, if you are recording in Analog mode, press the GET TEXT FROM SCREEN AREA softkey.

WinRunner is minimized 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. Press and hold down the left mouse button. Drag the mouse until the rectangle encompasses the entire text, then release the mouse button. Press the right mouse button to capture the string.

You can preview the string before you capture it. Press 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 428 for more information.

Books Online

Find

Find Again

Help

Top of Chapter

Back

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. For information about programming functions into your test scripts, see Chapter 20, **Generating Functions**. For information about specific functions, refer to the *TSL Online Reference.*

Books Online

Find

Find Again

Help

Top of Chapter

Back

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$_2$* specify the x- and y-coordinates of the upper left and bottom right corners of 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.

Books Online

Find

Find Again

Help

Top of Chapter

Back

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 search 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 x- and y-coordinates of the upper left and bottom right corners of 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,* [*,x₁,y₁,x₂,y₂*] [*,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 online 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 about 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);
```

Books Online

Find

Find Again

Help

Top of Chapter

Back

Teaching Fonts to WinRunner

You use the Fonts Expert 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 folder*/ *fonts* subfolder.

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 **Tools** > **Fonts Expert** or choose **Start** > **Programs** > **WinRunner** > **Fonts Expert**. The Fonts Expert opens.

2 Choose **Font** > **Learn**. The Learn Font dialog box opens.
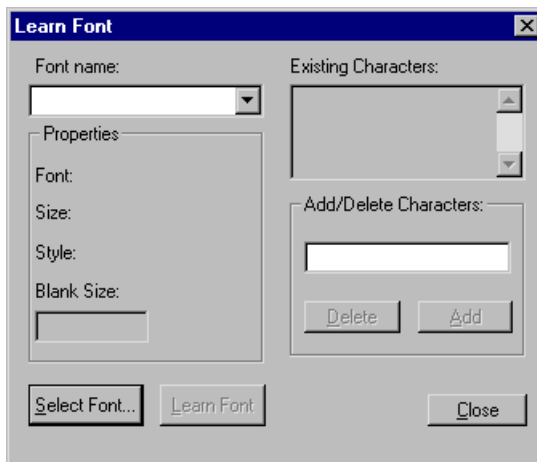
3 Type in a name for the new font in the **Font Name** box (maximum of eight characters, no extension).

4 Click **Select Font**. The Font dialog box opens.

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 **Font** > **Groups**. The Font Groups dialog box opens.
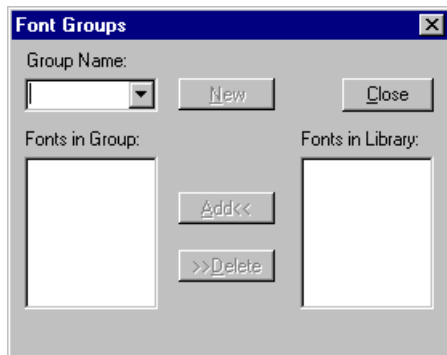
2 Type in a unique name in the **Group Name** box (up to eight characters, no extension).

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 **Font** > **Groups**. 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 **Font** > **Groups**. 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 **Settings > General Options**.

The General 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 General Options dialog box, see Chapter 36, **Setting Global Testing Options**. For more information about using the **setvar** function to choose a font group from within a test script, see Chapter 37, **Setting Testing Options from a Test Script**.

Books Online

Find

Find Again

Help

Top of Chapter

Back

Creating Tests
Creating Data-Driven Tests

WinRunner enables you to create and run tests which are driven by data stored in an external table.

This chapter describes:

- **The Data-Driven Testing Process**
- **Parameterizing a Data-Driven Test**
- **Guidelines for Creating a Data-Driven Test**
- **Importing Data from a Database**
- **Technical Specifications for the Data Table**
- **Editing the Data Table**
- **Data-Driven Checkpoints and Bitmap Synchronization Points**
- **Using TSL Functions with Data-Driven Tests**

Books Online

Find

Find Again

Help

Top of Chapter

Back

About Creating Data-Driven Tests

When you test your application, you may want to check how it performs the same operations with multiple 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*. You can create data-driven tests using the DataDriver Wizard or by manually adding data-driven statements to your test scripts.

Books Online

Find

Find Again

? Help

Top of Chapter

Back

The Data-Driven Testing Process

For non-data-driven tests, the testing process is performed in three steps: creating a test; running the test; analyzing test results. When you create a data-driven test, you perform an extra step between creating the test and running it: parameterizing the test.

The following diagram outlines the stages of the data-driven testing process in WinRunner:
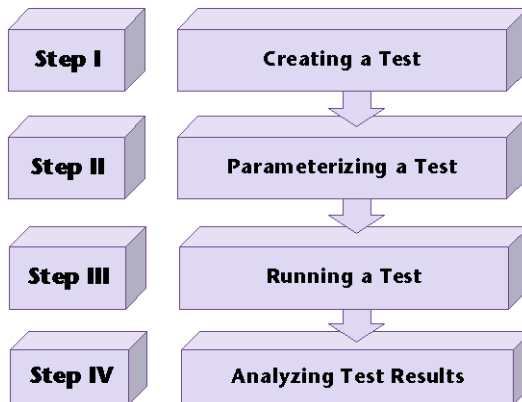
Step I - Creating a Test

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 checkpoints when you run the test. For information about adding checkpoints to your test scripts, see Chapter 9, **Checking GUI Objects**, Chapter 12, **Checking Table Contents**, Chapter 13, **Checking Databases**, Chapter 14, **Checking Bitmaps**, and Chapter 15, **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. You can use the DataDriver Wizard to parameterize your test or you can make these changes manually.

To create a data-driven test, you first record a test script and insert checkpoints. You then perform the following steps:

1 Create a data table containing variable values for the parameters. Each row in the table contains a set of data.

2 In selected recorded statements and checkpoints, replace fixed values with parameters.

3 Add statements to your test script to enable WinRunner to read from the data table and run in a loop for each set of data.

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. You replace a fixed values in your test script with a **ddt_val** statement, which returns the value of a parameter from an external data table.

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.

When you replace fixed values in your test script with parameters, you must also create a data table in which to store the variable data. When you run your test, WinRunner substitutes the parameters in your test script with values from the data table. During each iteration of your test, WinRunner changes the values in the parameterized statements.

Note: You can use Microsoft Query or Data Junction to import data from an external database into the data table. For more information, see **Importing Data from a Database** on page 471.

- 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 can 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 that open and close the data table

- control-flow statements that 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

  You can add these statements to your script automatically, using the DataDriver Wizard, or manually. For information on the DataDriver Wizard, see **Using the DataDriver Wizard** on page 447. For information on adding statements to your test script manually, see **Adding Statements to Your Test Script to Open and Close the Data Table and Run Your Test in a Loop** on page 461.

  You can also enhance the test with additional TSL statements. For example, you can change the active row in the data table or read data from the data table. For more information, see **Using TSL Functions with Data-Driven Tests** on page 502.

Step III - Running a Test

After you create a data-driven test, you run it as you would run any other WinRunner test. WinRunner substitutes the parameters in your test script with data from the data table. While WinRunner runs the test, it opens the data table. For each iteration of the test, it performs the operations you recorded on your application and conducts the checks you specified. For more information on running a test, see Chapter 27, **Running Tests**.

Note that if you chose to import data from a database, then when you run the test, the **ddt_update_from_db** function updates the data table with data from the database. For information on importing data from a database, see **Importing Data from a Database** on page 471. For information on the **ddt_update_from_db** function, see **Using TSL Functions with Data-Driven Tests** on page 502 or refer to the *TSL Online Reference*.

Books Online

M Find

Find Again

? Help

Top of Chapter

Back

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. The Test Results 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.

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 Log table in the Test Results window, identified as "Table Row" 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 511 or refer to the *TSL Online Reference*. For information on viewing test results, see Chapter 28, **Analyzing Test Results**.

Books Online

M Find

Find Again

? Help

Top of Chapter

← Back

Parameterizing a Data-Driven Test

Parameterizing a test, i.e., turning a basic WinRunner test into a data-driven test requires performing the following main steps:

1 Replace fixed values in checkpoint statements and in recorded statements with parameters, and create a data table containing values for the parameters.

2 Add 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.

3 Add statements to your script that open and close the data table.

4 Assign a variable name to the data table (mandatory when using the DataDriver Wizard and otherwise optional).

You can use the DataDriver Wizard to perform these steps, or you can modify your test script manually.
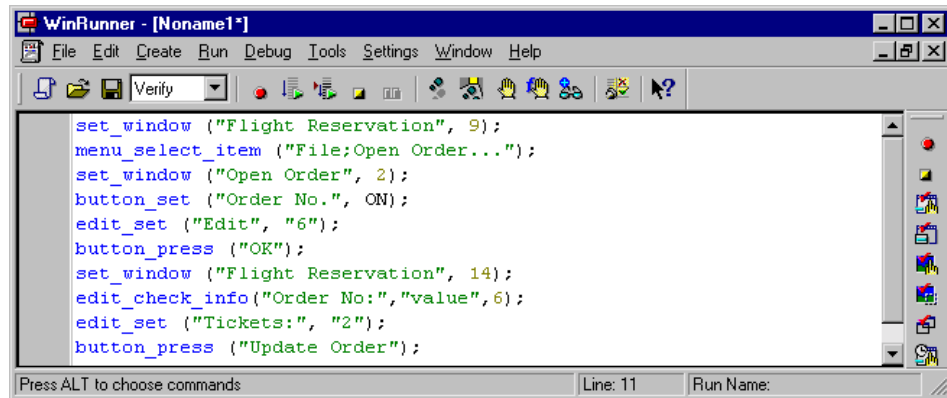
Creating a Basic Test

In order to create a data-driven test, you must first create a basic test and then parameterize it.

You create a basic test by recording a test, as usual, with one set of data. In the following example, the user wants to check that opening an order and updating the number of tickets in the order is performed correctly for a variety of orders. The test is recorded using one passenger's flight data.

To record this test, you open an order, use the **Create > GUI Checkpoint > For Single Property** command to check that the correct order is open. You change the number of tickets in the order and then update the order. A test script similar to the following is created:


```
set_window ("Flight Reservation", 9);
menu_select_item ("File;Open Order...");
set_window ("Open Order", 2);
button_set ("Order No.", ON);
edit_set ("Edit", "6");
button_press ("OK");
set_window ("Flight Reservation", 14);
edit_check_info("Order No:","value",6);
edit_set ("Tickets:", "2");
button_press ("Update Order");
```

The purpose of this test is to check that the correct order has been opened. Normally you would use the **Create > GUI Checkpoint > For Object/Window** command to insert an **obj_check_gui** statement in your test script. However, all **_check_gui** statements contain references to checklists, and since checklists do not contain fixed values, they cannot be parameterized from within a test script while creating a data-driven test. You have two options:

- As in the example above, you use the **Create > GUI Checkpoint > For Single Property** command to create a property check without a checklist. In this case, an **edit_check_info** statement checks the content of the edit field in which the order number is displayed. For information on checking a single property of an object, see Chapter 9, **Checking GUI Objects**. WinRunner can write an event to the Test Results window whenever these statements fail during a test run. To set this option, select the **Fail when single property check fails** check box in the Run tab of the General Options dialog box or use the **setvar** function to set the *single_prop_check_fail* testing option. For additional information, see Chapter 36, **Setting Global Testing Options**, or Chapter 37, **Setting Testing Options from a Test Script**.

You can use the **Create > GUI Checkpoint > For Single Property** command to create property checks using the following **_check_** functions:

button_check_info	**scroll_check_info**
edit_check_info	**static_check_info**
list_check_info	**win_check_info**
obj_check_info	

You can also use the following **_check** functions to check single properties of objects without creating a checklist. You can create statements with these functions manually or using the Function Generator. For additional information, see Chapter 20, **Generating Functions**.

button_check_state	**list_check_selected**
edit_check_selection	**scroll_check_pos**
edit_check_text	**static_check_text**
list_check_item	

For information about specific functions, refer to the *TSL Online Reference*.

- Alternatively, you can create data-driven GUI and bitmap checkpoints and bitmap synchronization points. For information on creating data-driven GUI and bitmap checkpoints and bitmap synchronization points, see **Data-Driven Checkpoints and Bitmap Synchronization Points** on page 493.

Books Online

Find

Find Again

Help

Top of Chapter

Back

Using the DataDriver Wizard

You can use the DataDriver Wizard to turn your entire script or a part of your script into a data-driven test. For example, your test script may include recorded operations, checkpoints, and other statements which do not need to be repeated for multiple sets of data. You need to parameterize only the portion of your test script that you want to run in a loop with multiple sets of data.

To create a data-driven test:

1 If you want to turn only part of your test script into a data-driven test, first select those lines in the test script.

Books Online

M Find

Find Again

? Help

Top of Chapter

⬅ Back

2 Choose **Tools > DataDriver Wizard**.

- If you selected part of the test script before opening the wizard, proceed to step 3.

- If you did not select any lines of script, the following screen opens:

If you want to turn only part of the test into a data-driven test, click **Cancel**. Select those lines in the test script and reopen the DataDriver Wizard.

If you want to turn the entire test into a data-driven test, click **Next**.
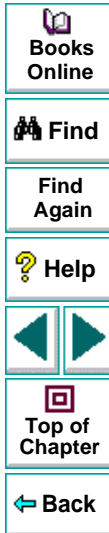
3 The following wizard screen opens:
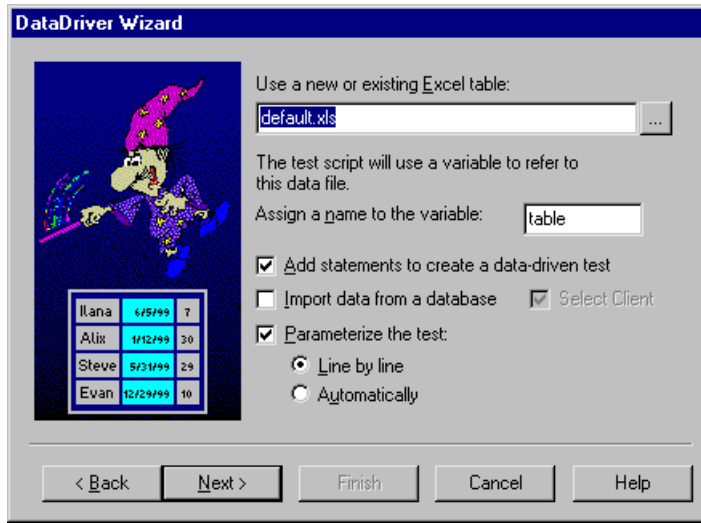
The **Use a new or existing Excel table** box displays the name of the Excel file that WinRunner creates, which stores the data for the data-driven test. You can accept the default data table for this test. Alternatively, you can enter a different name for the data table or use the browse button to locate the path of an existing data table. By default, the data table is stored in the test folder.

In the **Assign a name to the variable** box, enter a variable name with which to refer to the data table, or accept the default name, "table."
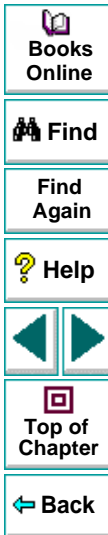
Choose from among the following options:

- **Add statements to create a data-driven test:** Sets a variable name by which to refer to the data table. Adds braces (**{** and **}**), a **for** statement, and a **ddt_get_row_count** statement to your test script selection to run it in a loop while it reads from the data table. Adds **ddt_open** and **ddt_close** statements to your test script to open and close the data table, which are necessary in order to iterate rows in the table. Note that you can also add these statements to your test script manually. For more information, see **Adding Statements to Your Test Script to Open and Close the Data Table and Run Your Test in a Loop** on page 461.

  If you do not choose this option, you will receive a warning that your data-driven test must contain a loop and statements to open and close your data table.

Note: You should not select this option if you have chosen it previously while running the DataDriver Wizard on the same portion of your test script.

- **Import data from a database:** Imports data from a database. This option adds **ddt_update_from_db**, and **ddt_save** statements to your test script after the **ddt_open** statement. For more information, see **Importing Data from a Database** on page 471.

Books Online

M Find

Find Again

? Help

Top of Chapter

← Back

Note that in order to import data from a database, either Microsoft Query or Data Junction must be installed on your machine. You can install Microsoft Query from the *custom installation* of Microsoft Office. Note that Data Junction is not automatically included in your WinRunner package. To purchase Data Junction, contact your Mercury Interactive representative. For detailed information on working with Data Junction, refer to the documentation in the Data Junction package.

Note: If the **Add statements to create a data-driven test** option is not selected along with the **Import data from a database** option, it also sets a variable name by which to refer to the data table. In addition, it adds **ddt_open** and **ddt_close** statements to your test script. Since there is no iteration in the test, the **ddt_close** statement is at the end of the block of **ddt_** statements, rather than at the end of the block of selected text.

● **Parameterize the test:** Replaces fixed values in selected checkpoints and in recorded statements with parameters, using the **ddt_val** function, and in the data table, adds columns with variable values for the parameters.

Line by line: Opens a wizard screen for each line of the selected test script, which enables you to decide whether to parameterize a particular line, and if so, whether to add a new column to the data table or use an existing column when parameterizing data.

Automatically: Replaces all data with **ddt_val** statements and adds new columns to the data table. The first argument of the function is the name of the column in the data table. The replaced data is inserted into the table.

Note: You can also parameterize your test manually. For more information, see **Creating a Data Table and Parameterizing Values in a Test Script** on page 462.

Note: The *ddt_func.ini* file in the *dat* folder lists the TSL functions that the DataDriver Wizard can parameterize while creating a data-driven test. This file also contains the index of the argument that by default can be parameterized for each function. You can modify this list to change the default argument that can be parameterized for a function. You can also modify this list to include user-defined functions or any other TSL functions, so that you can parameterize statements with these functions while creating a test. For information on creating user-defined functions, see Chapter 22, **Creating User-Defined Functions**.

Click **Next**.

Note that if you did not select any check boxes, only the Cancel button is enabled.

4 If you selected the **Import data from a database** check box in the previous screen, continue at **Importing Data from a Database** on page 471. Otherwise, the following wizard screen opens:

The **Test script line to parameterize** box displays the line of the test script to parameterize. The highlighted value can be replaced by a parameter.

The **Argument to be replaced** box displays the argument (value) that you can replace with a parameter. You can use the arrows to select a different argument to replace.
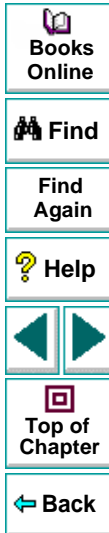
Choose whether and how to replace the selected data:

● **Do not replace this data:** Does not parameterize this data.

● **An existing column:** If parameters already exist in the data table for this test, select an existing parameter from the list.

● **A new column:** Creates a new column for this parameter in the data table for this test. Adds the selected data to this column of the data table. The default name for the new parameter is the logical name of the object in the selected TSL statement above. Accept this name or assign a new name.

In the sample Flight application test script shown earlier on **page 444**, there are several statements that contain fixed values entered by the user.

In this example, a new data table is used, so no parameters exist yet. In this example, for the first parameterized line in the test script, the user clicks the **Data from a new parameter** radio button. By default, the new parameter is the logical name of the object. You can modify this name. In the example, the name of the new parameter was modified to "Date of Flight."

The following line in the test script:

edit_set ("Edit", "6");

is replaced by:

edit_set("Edit",ddt_val(table,"Edit"));

The following line in the test script:

edit_check_info("Order No:","value",6);
is replaced by:

edit_check_info("Order No:","value",ddt_val(table,"Order_No"));

● To parameterize additional lines in your test script, click **Next**. The wizard displays the next line you can parameterize in the test script selection. Repeat the above step for each line in the test script selection that can be parameterized. If there are no more lines in the selection of your test script that can be parameterized, the final screen of the wizard opens.

● To proceed to the final screen of the wizard without parameterizing any additional lines in your test script selection, click **Skip**.

5 The final screen of the wizard opens.

● To perform the tasks specified in previous screens and close the wizard, click **Finish**.

● To close the wizard without making any changes to the test script, click **Cancel**.

Note: If you clicked **Cancel** after parameterizing your test script but before the final wizard screen, the data table will include the data you added to it. If you want to save the data in the data table, then open the data table and then save it.

Once you have finished running the DataDriver Wizard, the sample test script for the example on **page 444** is modified, as shown below:

Books Online

Find

Find Again

Help

Top of Chapter

Back

Statements to open data table and run test in a loop

Parameterized statement

Parameterized property check

End of loop

Statement to close data table

```
table = "default.xls";
rc = ddt_open(table, DDT_MODE_READ);
if (rc!= E_OK && rc != E_FILE_OPEN)
    pause("Cannot open table.");
ddt_get_row_count(table,table_RowCount);
for(table_Row = 1; table_Row <= table_RowCount; table_Row ++)
{
    ddt_set_row(table,table_Row);
    set_window ("Flight Reservation", 9);
    menu_select_item ("File;Open Order...");
    set_window ("Open Order", 2);
    button_set ("Order No.", ON);
    edit_set ("Edit",ddt_val(table,"Edit"));
    button_press ("OK");
    set_window ("Flight Reservation", 14);
    edit_check_info("Order No:","value",ddt_val(table,"Order_No"));
    edit_set ("Tickets:", "2");
    button_press ("Update Order");

}
ddt_close(table);
```

WinRunner - [Noname1*]

File Edit Create Run Debug Tools Settings Window Help

Ready   Line: 21   Run Name:

If you open the data table (**Tools > Data Table**), the **Open or Create a Data Table** dialog box opens. Select the data table you specified in the DataDriver Wizard. When the data table opens, you can see the entries made in the data table and edit the data in the table. For the previous example, the following entry is made in the data table.

	Edit	Order_No	C	D	E
1	6	6			
2					
3					
4					

Data Table - D:\Program Files\Mercury Interactive\WinR...
File Edit Data Format Help
A1   6
Ready

Manually Creating a Data-Driven Test

You can create a data-driven test manually, without using the DataDriver Wizard. Note that in order to create a data-driven test manually, you must complete all the steps described below:

- defining the data table

- add statements to your test script to open and close the data table and run your test in a loop

- import data from a database (optional)

- create a data table and parameterize values in your test script

Defining the Data Table

Add the following statement to your test script immediately preceding the parameterized portion of the script. This identifies the name and the path of your data table. Note that you can work with multiple data tables in a single test, and you can use a single data table in multiple tests. For additional information, see **Guidelines for Creating a Data-Driven Test** on page 469.

table="Default.xls";

Note that if your data table has a different name, substitute the correct name. By default, the data table is stored in the folder for the test. If you store your data table in a different location, you must include the path in the above statement.

For example:

table1 = "default.xls";

is a data table with the default name in the test folder.

table2 = "table.xls";

is a data table with a new name in the test folder.

table3 = "C:\\Data-Driven Tests\\Another Test\\default.xls";

is a data table with the default name and a new path. This data table is stored in the folder of another test.

Scripts created in WinRunner versions 5.0 and 5.01 may contain the following statement instead.

table=getvar("testname") & "\\Default.xls";

This statement is still valid. However, scripts created in WinRunner 6.0 use relative paths and therefore the full path is not required in the statement.

Adding Statements to Your Test Script to Open and Close the Data Table and Run Your Test in a Loop

Add the following statements to your test script immediately following the table declaration.

```
rc=ddt_open (table);
if (rc!= E_OK && rc != E_FILE_OPEN)
    pause("Cannot open table.");
ddt_get_row_count(table,table_RowCount);
for(table_Row = 1; table_Row <= table_RowCount ;table_Row ++ )
{
    ddt_set_row(table,table_Row);
```

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:

```
}
ddt_close (table);
```

These statements run the statements that appear within the curly brackets above for every 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.

Books Online

Find

Find Again

Help

Top of Chapter

Back

Importing Data from a Database

You must add **ddt_update_from_db**, and **ddt_save** statements to your test script after the **ddt_open** statement. You must use Microsoft Query to define a query in order to specify the data to import. For more information, see **Importing Data from a Database** on page 471. For more information on the **ddt_** functions, see **Using TSL Functions with Data-Driven Tests** on page 502 or refer to the *TSL Online Reference*.

Creating a Data Table and Parameterizing Values in a Test Script

You create a data-driven test by creating a data table in which data is stored and parameterizing fixed values within the test. You can open and save data tables other than the *default.xls* data table. This enables you to use several different data tables in one test script, if desired. You can use the **New**, **Open**, **Save**, and **Save As** commands in the data table to open and save data tables. For additional information, see **Editing the Data Table** on page 485.

In the sample test script in **Creating a Basic Test** on page 444, there are several statements that contain fixed values entered by the user:

edit_set("Edit", "6");

edit_check_info("Order No:","value",6);

You use the **Parameterize Data** dialog box to parameterize the statements and replace the data with parameters.

To replace the data in your test script with parameters and create a data table:

1 Choose **Tools > Data Table**. The **Open or Create a Data Table** dialog box opens. Select the data table you specified in the test script. The data table opens or is created.

To open an existing data table, choose **File > Open** from within the data table.

Note that by default, the name of the data table is *default.xls*. By default, 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: "6".

3 Choose **Tools > Parameterize Data**. The Parameterize Data dialog box opens.
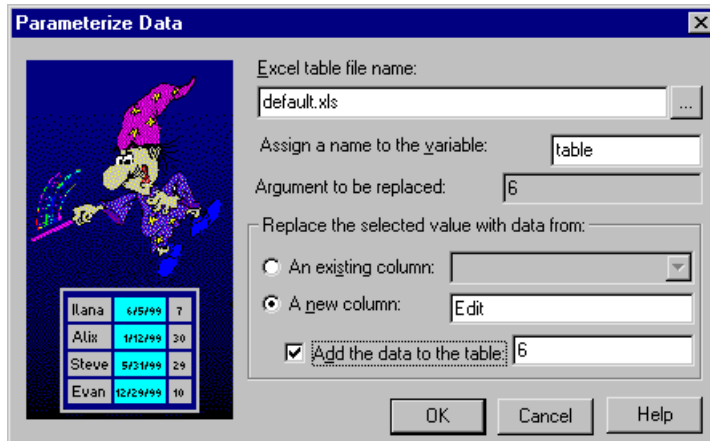
4 In the **Excel table file name box**, you can accept the default name and location of the data table, enter the different name for the data table, or use the browse button to locate the path of a data table. Note that by default the name of the data table is *default.xls*, and it is stored in the test folder. However, if you previously worked with a different data table in this test, then it appears here instead.

Click **A new column**. WinRunner suggests a name for the parameter in the box. You can accept this name or choose a different name. WinRunner creates a column with the same name as the parameter in the data table.

The data that was selected in your test script, with quotation marks, appears in the **Add the data to the table** box.

- If you want to include the data currently selected in the test script in the data table, select the **Add the data to the table** check box. You can modify the data in this box.

- If you do not want to include the data currently selected in the test script in the data table, clear the **Add the data to the table** check box.

- You can also assign the data to an existing parameter, which assigns the data to a column already in the data table. If you want to use an existing parameter, click **An existing column**, and select an existing column from the list.

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, with a corresponding column in the data table.

In the example, the value "6" is replaced with a **ddt_val** statement which contains the name of the table and the parameter "Edit", so that the original statement appears as follows:

edit_set ("Edit",ddt_val(table,"Edit"));

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 Edit.

Repeat steps 2 to 5 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 5.0/95 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.

	Edit	Order_No	C	D	E	
1	6	6				
2	4	4				
3	3	3				
4						

A1   6

Ready

- 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.

Books Online

Find

Find Again

Help

Top of Chapter

Back

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.

Note: Data tables support only Microsoft Excel 5.0/95. If you are working with Microsoft Excel 97 or Excel 2000, then before closing the data table, click **File > Save As**, and save the data table in Excel 5.0/95 format.

For more information on the **ddt_val** function, see **Using TSL Functions with Data-Driven Tests** on page 502 or refer to the *TSL Online Reference*.

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.

- You can open and save data tables other than the *default.xls* data table. This enables you to use several different data tables in one test script. You can use the **New**, **Open**, **Save**, and **Save As** commands in the data table to open and save data tables. For additional information, see **Editing the Data Table** on page 485.

    Note: If you open a data table from one test while it is open from another test, the changes you make to the data table in one test will not be reflected in the other test. To save your changes to the data table, you should save and close the data table in one test before opening it in another test.

- 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 502.

- 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 502.

- 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 parameterize statements containing GUI checkpoints, bitmap checkpoints, and bitmap synchronization points. For more information, see **Data-Driven Checkpoints and Bitmap Synchronization Points** on page 493.

- You can parameterize constants as you would any other string or value.

- You can use the data table in the same way as an Excel spreadsheet, including inserting formulas into cells.

- It is not necessary for the data table viewer to be open when you run a test.

- You can use the **ddt_set_val** and **ddt_set_val_by_row** functions to insert data into the data table during a test run. Then use the **ddt_save** function to save your changes to the data table.

Note: By default, the data table is stored in the test folder.

Importing Data from a Database

In order to import data from a database into a data table, you must specify the data to import. If you selected the **Import data from a database** check box, the DataDriver Wizard prompts you to specify the data to import. You can use either ODBC/Microsoft Query or Data Junction to determine how to import data from a database and to decide which data to import.

Note that in order to import data from a database, Microsoft Query or Data Junction must be installed on your machine. You can install Microsoft Query from the *custom installation* of Microsoft Office. Note that Data Junction is not automatically included in your WinRunner package. To purchase Data Junction, contact your Mercury Interactive representative. For detailed information on working with Data Junction, refer to the documentation in the Data Junction package.

Note: If you chose to replace data with data from an existing parameter in the database and you are importing data from a database, all column headers in your query appear in the list of existing parameters. If there is already a column with the same header in the data table, then the data in that column is automatically updated from the database. The data from the database overrides the data in the relevant column in the data table for all rows that are imported from the database.

Importing Data from a Database Using Microsoft Query

You can use Microsoft Query to choose a data source and define a query within the data source.

Note that WinRunner supports the following versions of Microsoft Query:

- version 2.00 (part of Microsoft Office 95)
- version 8.00 (part of Microsoft Office 97)
- version 2000 (part of Microsoft Office 2000)

Setting Microsoft Query Options

If Microsoft Query is installed on your machine, the following wizard screen opens once you choose to import data from a database:
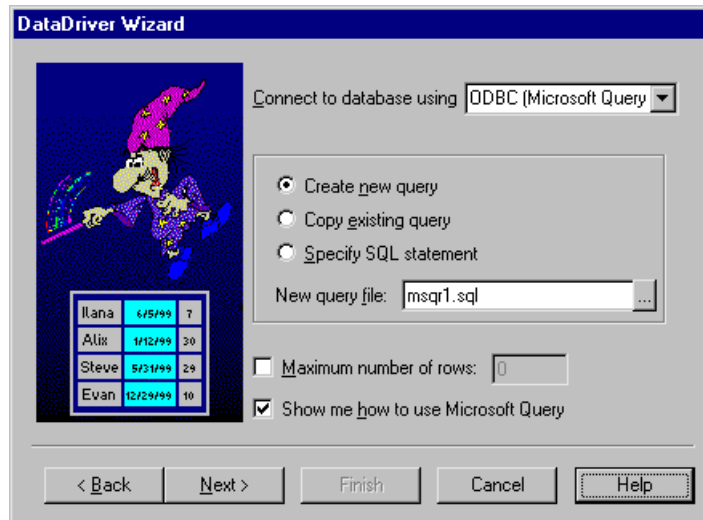
You can choose from the following options:

● **Create new query:** Opens Microsoft Query, enabling you to create a new ODBC *.sql* query file with the name specified below. For additional information, see **Creating a New Source Query File** on page 475.

● **Copy existing query:** Opens the **Select source query file** screen in the wizard, which enables you to copy an existing ODBC query from another query file. For additional information, see **Selecting a Source Query File** on page 476.

● **Specify SQL statement:** Opens the **Specify SQL statement** screen in the wizard, which enables you to specify the connection string and an SQL statement. For additional information, see **Specifying an SQL Statement** on page 477.

● **New query file:** Displays the default name of the new *.sql* query file for the data to import from the database. You can use the browse button to browse for a different *.sql* query file.

● **Maximum number of rows:** Select this check box and enter the maximum number of database rows to import. If this check box is cleared, there is no maximum. Note that this option adds an additional parameter to your **db_check** statement. For more information, refer to the *TSL Online Reference.*

● **Show me how to use Microsoft Query:** Displays an instruction screen.

Creating a New Source Query File

1 Choose a new or an existing data source.

2 Define a query.

3 When you are done:

- In version 2.00, choose **File > Exit and return to WinRunner** to close Microsoft Query and return to WinRunner.

- In version 8.00, in the Finish screen of the Query Wizard, click **Exit and return to WinRunner** and click **Finish** to exit Microsoft Query. Alternatively, click **View data or edit query in Microsoft Query** and click **Finish**. After viewing or editing the data, choose **File > Exit and return to WinRunner** to close Microsoft Query and return to WinRunner.

Once you finish defining your query, you return to the DataDriver Wizard to finish creating your data-driven test. For additional information, see step 4 on **page 453**.

Books Online

🔍 Find

Find Again

❓ Help

◀ ▶

🔲 Top of Chapter

⬅ Back

Selecting a Source Query File

The following screen opens if you chose to use an existing query file to import data from a database.

Books Online

Find

Find Again

Help

Top of Chapter

Back

Enter the pathname of the query file or use the **Browse** button to locate it. Once a query file is selected, you can use the **View** button to open the file for viewing.

Once you are done, you click **Next** to finish creating your data-driven test. For additional information, see step 4 on **page 453**.
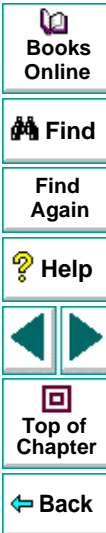
Specifying an SQL Statement

The following screen opens if you chose to specify an SQL statement to use to import data from a database into the data table.
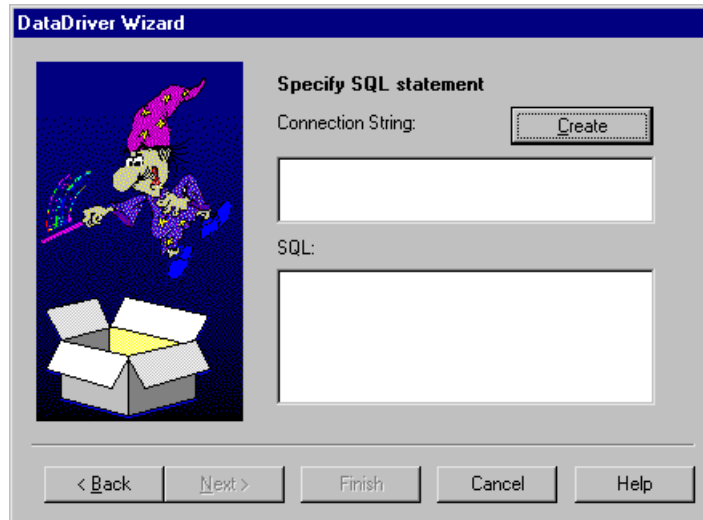
In this screen you must specify the connection string and the SQL statement:

- **Connection String:** Enter the connection string, or click **Create** to open the ODBC Select Data Source dialog box, in which you can select a **.dsn* file, which inserts the connection string in the box.

- **SQL:** Enter the SQL statement.

Once you are done, you click **Next** to finish creating your data-driven test. For additional information, see step 4 on **page 453**.

Once you import data from a database using Microsoft Query, the query information is saved in a query file called *msqrN.sql* (where N is a unique number). By default, this file is stored in the test folder (where the default data table is stored). The DataDriver Wizard inserts a **ddt_update_from_db** statement using a relative path and not a full path. During the test run, when a relative path is specified, WinRunner looks for the query file in the test folder. If the full path is specified for a query file in the **ddt_update_from_db** statement, then WinRunner uses the full path to find the location of the query file.

For additional information on using Microsoft Query, refer to the Microsoft Query documentation.

Importing Data from a Database Using Data Junction

You can use Data Junction to create a conversion file that converts a database to a target text file.

Note that WinRunner supports the versions 6.5 and 7 of Data Junction:

Setting Data Junction Options

If Data Junction is installed on your machine, the following wizard screen opens once you choose to import data from a database:
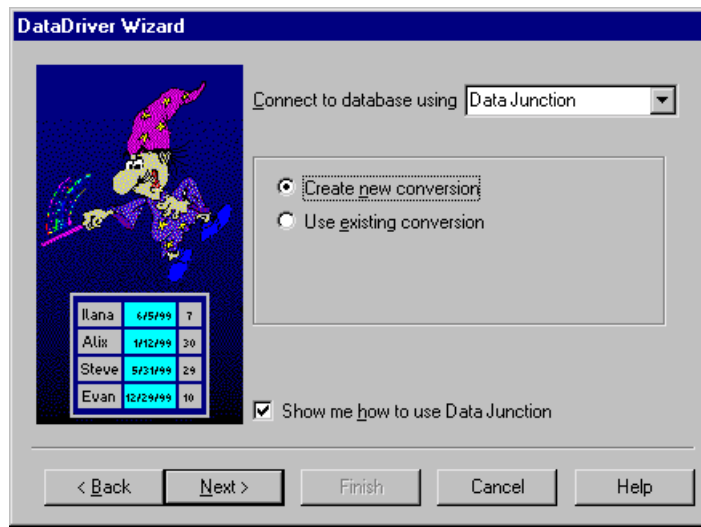
You can choose from the following options:

- **Create new conversion:** Opens Data Junction and enables you to create a new conversion file. For additional information, see **Creating a Conversion File in Data Junction** on page 481.

- **Use existing conversion:** Opens the **Select conversion file** screen in the wizard, which enables you to specify an existing conversion file. For additional information, see **Selecting a Data Junction Conversion File** on page 482.

- **Show me how to use Data Junction** (available only when **Create new conversion** is selected): Displays instructions for working with Data Junction.

Creating a Conversion File in Data Junction

  1 Specify and connect to the source database.

  2 Select an ASCII (delimited) target spoke type and specify and connect to the target file. Choose the "Replace File/Table" output mode.

Books Online

Find

Find Again

Note: If you are working with Data Junction version 7.0 and your source database includes values with delimiters (CR, LF, tab), then in the Target Properties dialog box, you must specify "\r\n\t" as the value for the **TransliterationIn** property. The value for the **TransliterationOut** property must be blank.

Help

  3 Map the source file to the target file.

  4 When you are done, click **File > Export Conversion** to export the conversion to a **.djs* conversion file.

Top of Chapter

Back

  5 The DataDriver Wizard opens to the **Select conversion file** screen. Follow the instructions in **Selecting a Data Junction Conversion File** on page 482.

For additional information on working with Data Junction, refer to the Data Junction documentation.

Selecting a Data Junction Conversion File

The following wizard screen opens when you are working with Data Junction.

Enter the pathname of the conversion file or use the **Browse** button to locate it. Once a conversion file is selected, you can use the **View** button to open the file for viewing.

You can also choose from the following options:

● **Copy conversion to test folder:** Copies the specified conversion file to the test folder.

● **Maximum number of rows:** Select this check box and enter the maximum number of database rows to import. If this check box is cleared, there is no maximum.

Once you are done, you click **Next** to finish creating your data-driven test. For additional information, see step 4 on **page 453**.
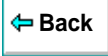
Technical Specifications for the Data Table

The following table displays the technical specifications for a data table.

maximum number of columns	256
maximum number of rows	16,384
maximum column width	255 characters
maximum row height	409 points
maximum formula length	1024 characters
number precision	15 digits
largest positive number	9.99999999999999E307
largest negative number	-9.99999999999999E307
smallest positive number	1E-307
smallest negative number	-1E-307
table format	Tabbed text file or Microsoft Excel 5.0/95 file. If you are using later versions of Microsoft Excel, use the Save As command to save your file in Microsoft Excel 5.0/95 version.
valid column names	Columns names must not include spaces. They can include only letters, numbers, and underscores (_).

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 also insert Excel formulas and functions into cells.

Note: If you do not want the data table editor to reformat your data (e.g. change the format of dates), then strings you enter in the data table should start with a quotation mark (**'**). This instructs the editor not to reformat the string in the cell.

To edit the data table:

1 Open your test.

2 Choose **Tools > Data Table**. The **Open or Create a Data Table** dialog box opens.

Books Online

M Find

Find Again

Help

Top of Chapter

Back

3 Select a data table for your test.

The data table for the test opens.
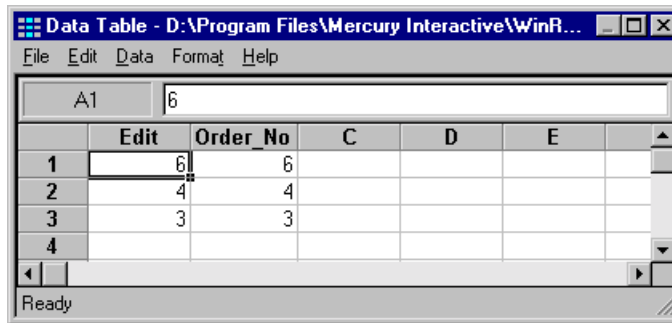
- Each row in the data table generally represents the values that WinRunner submits for all the parameterized fields during a single iteration of the test. A loop in a test that is associated with a table that has, for example, ten rows, will run ten times. TSL functions enable you to change the active 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 502, 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.

4 Use the menu commands described below to edit the data table.

5 Select **File > Close** to close the data table.
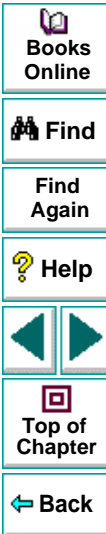
File Menu

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
New	Creates a new data table.
Open	Opens an existing data table. If you open a data table that was already opened by the **ddt_open** function, you are prompted to save and close it before opening it in the data table editor.
Save	Saves the active data table with its existing name and location.
Save As	Opens the Save As dialog box, which enables you to specify the name and location under which to save the data table.
Import	Imports an existing table file into the data table. This can be a Microsoft Excel 5.0/95 file or a tabbed text file. If you open a file that was already opened by the **ddt_open** function, you are prompted to save and close it before opening it in the data table editor. 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.

File Command	Description
Export	Saves the data table as a Microsoft Excel 5.0/95 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	Enables you to select the printer, the page orientation, and paper size.

Books Online

Find

Find Again

Help

Top of Chapter

Back

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 and 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.

Books Online

Find

Find Again

Help

Top of Chapter

Back

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 a specified value. 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 a specified value and replaces it with a different value. 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 mm/dd/yy format.
h:mm AM/PM	Sets format to Time with the H:MM AM/PM format.
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.

Books Online

Find

Find Again

Help

Top of Chapter

Back

Data-Driven Checkpoints and Bitmap Synchronization Points

When you create a data-driven test, you parameterize fixed values in TSL statements. However, GUI and bitmap checkpoints and bitmap synchronization points do not contain fixed values. Instead, these statements contain the following:

● A GUI checkpoint statement (**obj_check_gui** or **win_check_gui**) contains references to a checklist stored in the test's *chklist* folder and expected results stored in the test's *exp* folder.

● A bitmap checkpoint statement (**obj_check_bitmap** or **win_check_bitmap**) or a bitmap synchronization point statement (**obj_wait_bitmap** or **win_wait_bitmap**) contains a reference to a bitmap stored in the test's *exp* folder.

Note: When you check properties of GUI objects in a data-driven test, it is better to create a single property check than to create a GUI checkpoint: A single property check does not contain checklist, so it can be easily parameterized. You use the **Create > GUI Checkpoint > For Single Property** command to create a property check without a checklist. For additional information on using single property checks in a data-driven test, see **Creating a Basic Test** on page 444. For information on checking a single property of an object, see Chapter 9, **Checking GUI Objects**.

In order to parameterize GUI and bitmap checkpoints and bitmap synchronization points statements, you insert dummy values into the data table for each expected results reference. First you create separate columns for each checkpoint or bitmap synchronization point. Then you enter dummy values in the columns to represent captured expected results. Each dummy value should have a unique name (for example, gui_exp1, gui_exp2, etc.). When you run the test in Update mode, WinRunner captures expected results during each iteration of the test (i.e. for each row in the data table) and saves all the results in the test's *exp* folder.

● For a GUI checkpoint statement, WinRunner captures the expected values of the object properties.

● For a bitmap checkpoint statement or a bitmap synchronization point statement, WinRunner captures a bitmap.

To create a data-driven checkpoint or bitmap synchronization point:

 1 Create the initial test by recording or programming.

In the example below, the recorded test opens the Search dialog box in the Notepad application, searches for a text and checks that the appropriate message appears. Note that a GUI checkpoint, a bitmap checkpoint, and a synchronization point are all used in the example.

```
set_window ("Untitled - Notepad", 12);
menu_select_item ("Search;Find...");
set_window ("Find", 5);
edit_set ("Find what:", "John");
button_press ("Find Next");
set_window("Notepad", 10);
obj_check_gui("Message", "list1.ckl", "gui1", 1);
win_check_bitmap("Notepad", "img1", 5, 30, 23, 126, 45);
obj_wait_bitmap("Message", "img2", 13);
set_window ("Notepad", 5);
button_press ("OK");
set_window ("Find", 4);
button_press ("Cancel");
```

Books Online

M Find

Find Again

? Help

Top of Chapter

Back

2 Use the DataDriver Wizard (**Tools > DataDriver Wizard**) to turn your script into a data-driven test and parameterize the data values in the statements in the test script. For additional information, see **Using the DataDriver Wizard** on page 447. Alternatively, you can make these changes to the test script manually. For additional information, see **Manually Creating a Data-Driven Test** on page 458.

In the example below, the data-driven test searches for several different strings. WinRunner reads all these strings from the data table.

```
set_window ("Untitled - Notepad", 12);
menu_select_item ("Search;Find...");
table = "default.xls";
rc = ddt_open(table, DDT_MODE_READ);
if (rc!= E_OK && rc != E_FILE_OPEN)
    pause("Cannot open table.");
ddt_get_row_count(table,RowCount);
for (i = 1; i <= RowCount; i++) {
    ddt_set_row(table,i);
    set_window ("Find", 5);
    edit_set ("Find what:", ddt_val(table, "Str"));
    button_press ("Find Next");
    set_window("Notepad", 10);
```

 # The GUI checkpoint statement is not yet parameterized.
obj_check_gui("message", "list1.ckl", "gui1", 1);

 # The bitmap checkpoint statement is not yet parameterized.
win_check_bitmap("Notepad", "img1", 5, 30, 23, 126, 45);

 # The synchronization point statement is not yet parameterized.
obj_wait_bitmap("message", "img2", 13);
set_window ("Notepad", 5);
button_press ("OK");
}
ddt_close(table);
set_window ("Find", 4);
button_press ("Cancel");

For example, the data table might look like this:

	Str	B	C	D	E
1	John				
2	Susan				
3	Bill				
4					

Data Table - D:\Program Files\Mercury Interactive\...
File Edit Data Format Help
A3   Bill
Ready

Note that the GUI and bitmap checkpoints and the synchronization point in this data-driven test will fail on the 2nd and 3rd iteration of the test run. The checkpoints and the synchronization point will fail because the values for these points were captured using the "John" string, in the original recorded test. Therefore, they will not match the other strings taken from the data table.

3 Create a column in the data table for each checkpoint or synchronization point to be parameterized. For each row in the column, enter dummy values. Each dummy value should be unique.

For example, the data table in the previous step might now look like this:

	Str	GUI_Check1	BMP_Check1	Sync1
1	John	gui_exp1	bmp_exp1	sync_exp1
2	Susan	gui_exp2	bmp_exp2	sync_exp2
3	Bill	gui_exp3	bmp_exp3	sync_exp3
4				

Data Table - D:\Program Files\Mercury Interactive\...
File Edit Data Format Help
A4
Ready

4 Choose **Tools > Parameterize Value** to open the Assign Parameter dialog box. In the **Existing Parameter** box, change the expected values of each checkpoint and synchronization point to use the values from the data table. For additional information, see **Creating a Data Table and Parameterizing Values in a Test Script** on page 462. Alternatively, you can edit the test script manually.

For example, the sample script will now look like this:

```
set_window ("Untitled - Notepad", 12);
menu_select_item ("Search;Find...");
table = "default.xls";
rc = ddt_open(table, DDT_MODE_READ);
if (rc!= E_OK && rc != E_FILE_OPEN)
    pause("Cannot open table.");
ddt_get_row_count(table,RowCount);
for (i = 1; i <= RowCount; i++) {
    ddt_set_row(table,i);
    set_window ("Find", 5);
    edit_set ("Find what:", ddt_val(table, "Str"));
    button_press ("Find Next");
    set_window("Notepad", 10);
```

```
# The GUI checkpoint statement is now parameterized.
obj_check_gui("message", "list1.ckl",
              ddt_val(table, "GUI_Check1"), 1);

# The bitmap checkpoint statement is now parameterized.
win_check_bitmap("Notepad",
              ddt_val(table, "BMP_Check1"), 5, 30, 23, 126, 45);

# The synchronization point statement is now parameterized.
obj_wait_bitmap("message",
            ddt_val(table, "Sync1"), 13);
set_window ("Notepad", 5);
button_press ("OK");
}
ddt_close(table);
set_window ("Find", 4);
button_press ("Cancel");
```

5 Select **Update** in the run mode box to run your test in Update mode. Choose a **Run** command to run your test.

While the test runs in Update mode, WinRunner reads the names of the expected values from the data table. Since WinRunner cannot find the expected values for GUI checkpoints, bitmaps checkpoints, and bitmap synchronization points in the data table, it recaptures these values from your application and save them as expected results in the *exp* folder for your test. Expected values for GUI checkpoints are saved as expected results. Expected values for bitmap checkpoints and bitmap synchronization points are saved as bitmaps.

Once you have run your test in Update mode, all the expected values for all the sets of data in the data table are recaptured and saved.

Later you can run your test in Verify mode to check the behavior of your application.

Note: When you run your test in Update mode, WinRunner recaptures expected values for GUI and bitmap checkpoints automatically. WinRunner prompts you before recapturing expected values for bitmap synchronization points.

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 20, **Generating Functions**. For more information about these functions, refer to the *TSL Online Reference*.

Note: You must use the **ddt_open** function to open the data table before you use any other **ddt_** functions.

Opening a Data Table

The **ddt_open** function creates or opens the specified data table. The data table is a Microsoft Excel 5.0/95 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, mode* **);**

The *data_table_filename* is the name of the data table file. The *mode* is the mode for opening the data table: DDT_MODE_READ (read-only) or DDT_MODE_READWRITE (read or write).

Saving a Data Table

The **ddt_save** function saves the information in the specified data table. This function has the following syntax:

ddt_save (*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.

Exporting a Data Table

The **ddt_export** function exports the information of one table file into a different table file. This function has the following syntax:

ddt_export (*data_table_filename1*, *data_table_filename2* **);**

The *data_table_filename1* is the name of the source data table file. The *data_table_filename2* is the name of the destination data table file.

Books Online

Find

Find Again

Help

Top of Chapter

Back

Displaying the Data Table Editor

The **ddt_show** function shows or hides the editor of a given data table. This function has the following syntax:

ddt_show (*data_table_filename* **[,** *show_flag* **]);**

The *data_table_filename* is the name of the table. The *show_flag* is the value indicating whether the editor should be displayed (default=1) or hidden (0).

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.

Setting a Value in the Current Row of the Table

The **ddt_set_val** function writes a value into the current row of the table. This function has the following syntax:

ddt_set_val (*data_table_filename***,** *parameter***,** *value* **);**

The *data_table_filename* is the name of the data table file. The *parameter* is the name of the column into which the value will be inserted. The *value* is the value to be written into the table.

Note: You can only use this function if the data table was opened in DDT_MODE_READWRITE (read or write mode).

Setting a Value in a Row of the Table

The **ddt_set_val_by_row** function sets a value in a specified row of the table. This function has the following syntax:

ddt_set_val_by_row (** *data_table_filename*, ** *row***, ** *parameter***, ** *value* **);**

The *data_table_filename* is the name of the data table file. The *row* is the row number in the table. It can be any existing row or the current row number plus 1, which will add a new row to the data table. The *parameter* is the name of the column into which the value will be inserted. The *value* is the value to be written into the table.

Note: You can only use this function if the data table was opened in DDT_MODE_READWRITE (read or write mode).

Books Online

Find

Find Again

Help

Top of Chapter

Back

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 a List of Parameters in a Data Table

The **ddt_get_parameters** function returns a list of all parameters in the specified data table. This function has the following syntax:

ddt_get_parameters (*data_table_filename*, *params_list*, *params_num* **);**

The *data_table_filename* is the name of the data table file. The *params_list* is the out parameter that returns the list of all parameters in the data table, separated by tabs. The *params_name* is the out parameter that returns the number of parameters in *params_list*.

Returning the Value of a Parameter in the Active Row in a Data Table

The **ddt_val** function returns the value of a parameter in the active row in the specified data table. 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 in a Row in a Data Table

The **ddt_val_by_row** function returns the value of a parameter in the specified row of the specified data table. 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.

Importing Data from a Database into a Data Table

The **ddt_update_from_db** function imports data from a database into a data table. It is inserted into your test script when you select the Import data from a database option in the DataDriver Wizard. When you run your test, this function updates the data table with data from the database. This function has the following syntax:

ddt_update_from_db (*data_table_filename***,** *file,out_row_count*
   [, *max_rows* **]);**

The *data_table_filename* is the name of the data table file.

The *file* is an **.sql* file containing a query defined by the user in Microsoft Query or **.djs* file containing a conversion defined by Data Junction. The *out_row_count* is an out parameter containing the number of rows retrieved from the data table. The *max_rows* is an in parameter specifying the maximum number of rows to be retrieved from a database. If no maximum is specified, then by default the number of rows is not limited.

Note: You must use the **ddt_open** function to open the data table in DDT_MODE_READWRITE (read or write) mode. After using the **ddt_update_from_db** function, the new contents of the table are not automatically saved. To save the new contents of the table, use the **ddt_save** function before the **ddt_close** function.

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 cue before continuing the test.

This chapter describes:

● **Waiting for Property Values of Objects and Windows**

● **Waiting for Bitmaps of Objects and Windows**

● **Waiting for Bitmaps of Screen Areas**

About Synchronizing the Test Run

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 that run over a network. A synchronization point in your test script tells WinRunner to suspend running the test until the application being tested is ready, and then to continue the test.

There are two kinds of synchronization points: property value synchronization points, and bitmap synchronization points.

● When you want WinRunner to wait for an object or a window to have a specified property, you create a property value synchronization point.

● When you want WinRunner to wait for a visual cue to be displayed, you create a bitmap synchronization point. In a bitmap synchronization point, WinRunner waits for the bitmap of an object, a window, or an area of the screen to appear.

For example, suppose that while 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 WinRunner to wait for the import command to be completed before rotating the bitmap.

In another example, suppose that while testing an application you want to check that a button is enabled. Suppose that in your application the button becomes enabled only after your application completes an operation over the network. The time it takes for the application to complete this network operation depends on the load on the network. A human user would know to wait until the operation is completed and the button is enabled before clicking it. WinRunner, however, requires a synchronization point after launching the network operation and before clicking the button. Each time the test is run, the synchronization point tells WinRunner to wait for the button to become enabled before clicking it.

You can synchronize your test to wait for a bitmap of a window or a GUI object in your application, or on any rectangular area of the screen. You can also synchronize your test to wait for a property value of a GUI object, such as "enabled," to appear. To create a synchronization point, you choose a Create > Synchronization Point command indicate an area or an object in the application being tested. Depending on which Synchronization Point command you choose, WinRunner either captures the property value of a GUI object or a bitmap of a GUI object or area of the screen, and stores it in the expected results folder (*exp*). You can also modify the property value of a GUI object that is captured before it is saved in the expected results folder.

A bitmap synchronization point is a synchronization point that captures a bitmap. It appears as a **win_wait_bitmap** or **obj_wait_bitmap** statement in the test script. A property value synchronization point is a synchronization point that captures a property value. It appears as a **_wait_info** statement in your test script, such as **button_wait_info** or **list_wait_info**. When you run the test, WinRunner suspends the test run and waits for the expected bitmap or property value to appear. It then compares the current *actual* bitmap or property value with the *expected* bitmap or property value saved earlier. When the bitmap or property value appears, the test continues.

Note that when recording a test in Analog mode, you should press the SYNCHRONIZE BITMAP OF OBJECT/WINDOW or the SYNCHRONIZE BITMAP OF SCREEN AREA softkey to create a bitmap 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*.

Note about data-driven testing: In order to use bitmap synchronization points in data-driven tests, you must parameterize the statements in your test script that contain them. For information on using bitmap synchronization points in data-driven tests, see **Data-Driven Checkpoints and Bitmap Synchronization Points** on page 493.

Books Online

Find

Find Again

Help

Top of Chapter

Back

Waiting for Property Values of Objects and Windows

You can create a property value synchronization point, which instructs WinRunner to wait for a specified property value to appear in a GUI object. For example, you can tell WinRunner to wait for a button to become enabled or for an item to be selected from a list.

The method for synchronizing a test is identical for property values of objects and windows. You start by choosing Create > Synchronization Point > For Object/Window Property. As you pass the mouse pointer over your application, objects and windows flash. To select a window, you click the title bar or the menu bar of the desired window. To select an object, you click the object.

A dialog box opens containing the name of the selected window or object. You can specify which property of the window or object to check, the expected value of that property, and the amount of time that WinRunner waits at the synchronization point.

A statement with one of the following functions is added to the test script, depending on which GUI object you selected:

GUI Object	TSL Function
button	**button_wait_info**
edit field	**edit_wait_info**
list	**list_wait_info**
menu	**menu_wait_info**
an object mapped to the generic "object" class	**obj_wait_info**
scroll bar	**scroll_wait_info**
spin box	**spin_wait_info**
static text	**static_wait_info**
status bar	**statusbar_wait_info**
tab	**tab_wait_info**
window	**win_wait_info**

During a test run, WinRunner suspends the test run until the specified property value in a GUI object is detected. It then compares the current value of the specified property with its expected value. If the property values match, then WinRunner continues the test.

In the event that the specified property value of the GUI object does not appear, WinRunner displays an error message, when the *mismatch_break* testing option is on. For information about the *mismatch_break* testing option, see Chapter 37, **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 General Options dialog box. For information about setting this testing option globally, see Chapter 36, **Setting Global Testing Options**.

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 26, **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 property value synchronization point:

1. Choose **Create > Synchronization Point > For Object/Window Property** or click the **Synchronization Point for Object/Window Property** button on the User toolbar. The mouse pointer becomes a pointing hand.

2. Highlight the desired object or window. To highlight an object, place the mouse pointer over it. To highlight a window, place the mouse pointer over the title bar or the menu bar.

3. Click the left mouse button. Depending on whether you clicked an object or a window, either the **Wait for Object** or the **Wait for Window** dialog box opens.
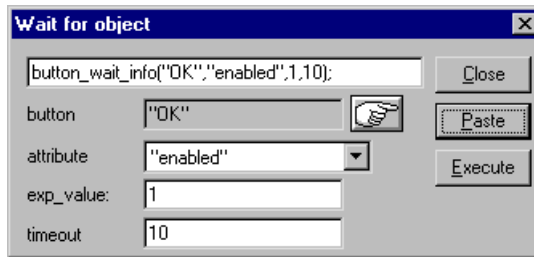
4 Specify the parameters of the property check to be carried out on the window or object, as follows:

- Window or type of object: The name of the window or object you clicked appears in a read-only box. To select a different window or object, click the pointing hand.

- Property: Select the property of the object or window to be checked from the list. The default property for the window or type of object specified above appears by default in this box.

- Expected value: Enter the expected value of the property of the object or window to be checked. The current value of this property appears by default in this box.

- Timeout: Enter the amount of time (in seconds) that WinRunner waits at the synchronization point in addition to the amount of time that WinRunner waits specified in the *timeout_msec* testing option. You can change the default amount of time that WinRunner waits using the *timeout_msec* testing option. For more information, see Chapter 37, **Setting Testing Options from a Test Script**. You can also change the default timeout value in the **Timeout for Checkpoints and CS Statements** box in the Run tab of the General Options dialog box. For more information, see Chapter 36, **Setting Global Testing Options**.

Note: Any changes you make to the above parameters appear immediately in the text box at the top of the dialog box.

Books Online

Find

Find Again

Help

Top of Chapter

Back

5 Click **Paste** to paste this statement into your test script.

The dialog box closes and a **_wait_info** statement that checks the property values of an object is inserted into your test script. 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 at the synchronization point, added to the *timeout_msec* testing option. For more information on **_wait_info** statements, refer to the *TSL Online Reference*.
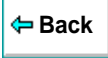
For example, suppose that while testing the Flight Reservation application you order a plane ticket by typing in passenger and flight information and clicking Insert. The application takes a few seconds to process the order. Once the operation is completed, you click Delete to delete the order.

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",1,"10");
button_press ("Delete");

Note: You can also use the Function Generator to create a synchronization point that waits for a property value of a window or an object. For information on using the Function Generator, see Chapter 20, **Generating Functions**. For more information about working with these functions, refer to the *TSL Online Reference*.

Waiting for Bitmaps of Objects and Windows

You can create a bitmap synchronization point that waits for the bitmap of an object or a window to appear in the application being tested.

The method for synchronizing a test is identical for bitmaps of objects and windows. You start by choosing Create > Synchronization Point > For Object/Window Bitmap. As you pass the mouse pointer over your application, objects and windows flash. To select the bitmap of an entire window, you click the window's title bar or menu bar. To select the bitmap of an object, you click the object.

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, then WinRunner continues the test.

In the event of a mismatch, WinRunner displays an error message, when the *mismatch_break* testing option is on. For information about the *mismatch_break* testing option, see Chapter 37, **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 General Options dialog box. For information about setting this testing option globally, see Chapter 36, **Setting Global Testing Options**.

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 26, **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 bitmap synchronization point for an object or a window:

1 Choose **Create > Synchronization Point > For Object/Window Bitmap** or click the **Synchronization Point for Object/Window Bitmap** on the User toolbar. Alternatively, if you are recording in Analog mode, press a SYNCHRONIZE BITMAP OF OBJECT/WINDOW softkey. The mouse pointer becomes a pointing hand.

2 Highlight the desired window or object. To highlight an object, place the mouse pointer over it. To highlight a window, place the mouse pointer over its title bar or menu bar.

3 Click the left mouse button to complete the operation. WinRunner captures the bitmap and generates an **obj_wait_bitmap** or a **win_wait_bitmap** statement with the following syntax in the test script.

obj_wait_bitmap (*object*, *image*, *time* **);**

win_wait_bitmap (*window*, *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 **obj_wait_bitmap** and **win_wait_bitmap** is affected by the current values specified for the *delay_msec*, *timeout_msec* and *min_diff* testing options. For more information on these testing options and how to modify them, see Chapter 37, **Setting Testing Options from a Test Script**. You may also set these testing options globally, using the corresponding **Delay for Window Synchronization**, **Timeout for Checkpoints and CS Statements**, and **Threshold for Difference between Bitmaps** boxes in the Run tab of the General Options dialog box. For more information about setting these testing options globally, see Chapter 36, **Setting Global Testing Options**.

Waiting for Bitmaps of Screen Areas

You can create a bitmap synchronization point that waits for a bitmap of a selected area in your application. You can define any rectangular area of the screen and capture it as a bitmap for a synchronization point.

You start by choosing Create > Synchronization Point > For Screen Area Bitmap. As you pass the mouse pointer over your application, it becomes a crosshairs pointer, and a help window opens in the top left corner of your screen.

You use the crosshairs mouse pointer to outline a rectangle around the area. The area can be any size: it can be part of a single window, or it can intersect several windows. 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 in which the area is located. 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 (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, then WinRunner continues the test.
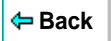
Books Online

Find

Find Again

? Help

Top of Chapter

Back

In the event of a mismatch, WinRunner displays an error message, when the *mismatch_break* testing option is on. For information about the *mismatch_break* testing option, see Chapter 37, **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 General Options dialog box. For information about setting this testing option globally, see Chapter 36, **Setting Global Testing Options**.

To define a bitmap synchronization point for an area of the screen:

 1 Choose **Create > Synchronization Point > For Screen Area Bitmap** or click the **Synchronization Point for Screen Area Bitmap** button on the User toolbar. Alternatively, if you are recording in Analog mode, press the SYNCHRONIZE BITMAP OF SCREEN AREA softkey.

   The WinRunner window is minimized to an icon, the mouse pointer becomes a crosshairs pointer, and a help window opens in the top left corner of your screen.

 2 Mark the area to be captured: press 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 bitmap and generates a **win_wait_bitmap** or **obj_wait_bitmap** statement with the following syntax in your test script.

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

   obj_wait_bitmap (*object*, *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.*

Books Online

M Find

Find Again

? Help

Note: The execution of **win_wait_bitmap** and **obj_wait_bitmap** statements is affected by the current values specified for the *delay_msec*, *timeout_msec* and *min_diff* testing options. For more information on these testing options and how to modify them, see Chapter 37, **Setting Testing Options from a Test Script**. You may also set these testing options globally, using the corresponding **Delay for Window Synchronization**, **Timeout for Checkpoints and CS Statements**, and **Threshold for Difference between Bitmaps** boxes in the Run tab of the General Options dialog box. For more information about setting these testing options globally, see Chapter 36, **Setting Global Testing Options**.

Top of Chapter

Back

Creating Tests
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:

- **Handling Pop-Up Exceptions**
- **Handling TSL Exceptions**
- **Handling Object Exceptions**
- **Activating and Deactivating Exception Handling**

Books Online

Find

Find Again

Help

Top of Chapter

Back

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.
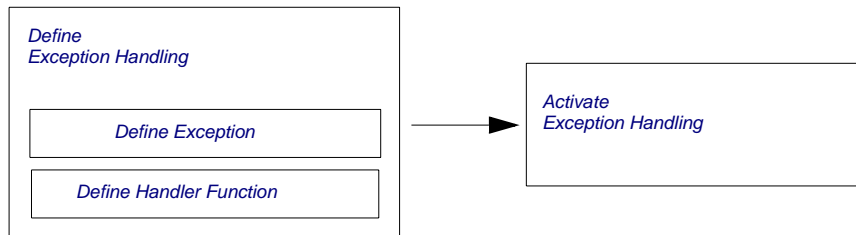
Define exception handling: Describe the event or error you want WinRunner to detect, and define the actions it will perform in order to resume test execution. To do this, you define the exception and define a handler function.

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.
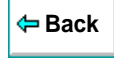
WinRunner enables you to handle the following types of exceptions:

● **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.

Note: When the WebTest add-in is loaded, you can instruct WinRunner to handle unexpected events and errors that occur in your Web site during a test run. For more information, refer to the *WebTest User's Guide.*

Handling Pop-Up Exceptions

A test run 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 **Tools > Exception Handling** to open the Exceptions dialog box.

Books Online

Find

Find Again

Help

Top of Chapter

Back
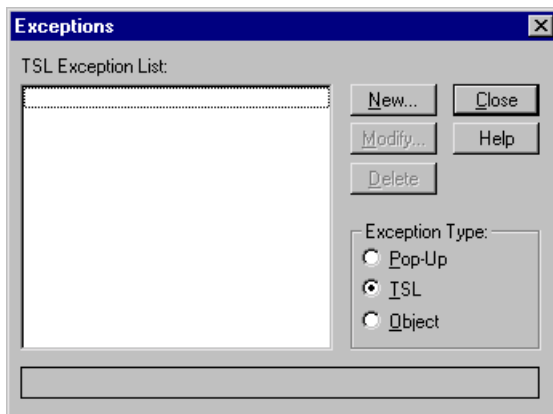
2 In the **Exception Type** box, click **Pop-Up**. Then click **New**.

The Pop-Up Exception dialog box opens.
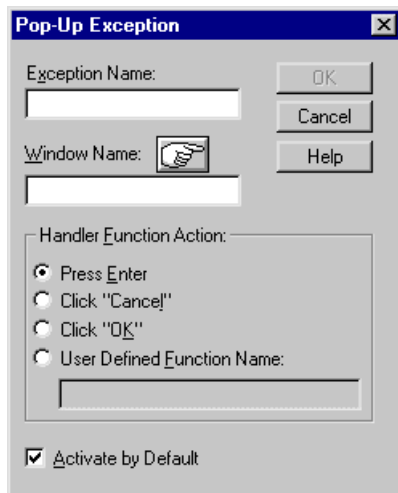
3 In the **Exception Name** box, type in a new name.

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 Enter, click Cancel, or click OK), or click **User Defined Function Name** to specify a user-defined handler. If you choose the last option, the dialog box changes to display the **User Defined Function Name** box.

If you specify a user-defined handler function in the **User Defined Function Name** box 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 539.

6 To activate exception handling active at all times, select the **Activate by Default** check box.
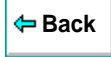
7 Click **OK** to complete the definition and close the dialog box. The new exception appears in the **Pop-Up Exception List** in the Exceptions dialog box.

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 557. 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.

Books Online

M Find

Find Again

? Help

Top of Chapter

← Back
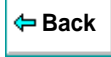
To define your own handler function using the Handler Function Definition dialog box:

 1 Define an exception using the **Pop-Up Exception** dialog box, as described in **Defining Pop-Up Exceptions** on page 535. 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:

```
public function close_window (in windo
{

}
```

Paste   Close   Help
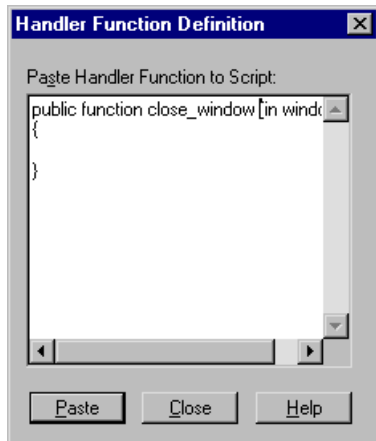
 3 Create a function that closes the window and recovers test execution.

 4 Click **Paste** to paste the statements into the WinRunner window.

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 23, **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);
    set_window(win_name);
    button_press ("OK");
}
```

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 **Tools** > **Exception Handling** to open the Exceptions dialog box.

2 In the **Exception Type** box, click **TSL**. Then click **New**.

The TSL Exception dialog box opens.
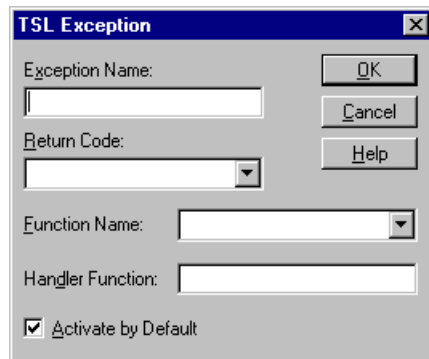
3 In the **Exception Name** box, type in a new name.

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 546.

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 557. 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*.

Books Online

Find

Find Again

Help

Top of Chapter

Back

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.
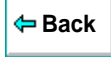
To define a handler function using the Handler Function Definition dialog box:

1 Define an exception using the **TSL Exception** dialog box, as described in **Defining TSL Exceptions** on page 544. 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.

5 Click **Close** to close 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.

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 23, **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.

Books Online

Find

Find Again

Help

Top of Chapter

Back

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 application you are testing. These changes are often subtle but they can disrupt the test run 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.

Books Online

Find

Find Again

Help

Top of Chapter

Back

Defining Object Exceptions

You use the Object Exception dialog box to define, modify, and delete object exceptions.

To define an object exception:

1 Choose **Tools** > **Exception Handling** to open the Exceptions dialog box.

2 In the **Exception Type** box, click **Object**. Then click **New**.

The Object Exception dialog box opens.
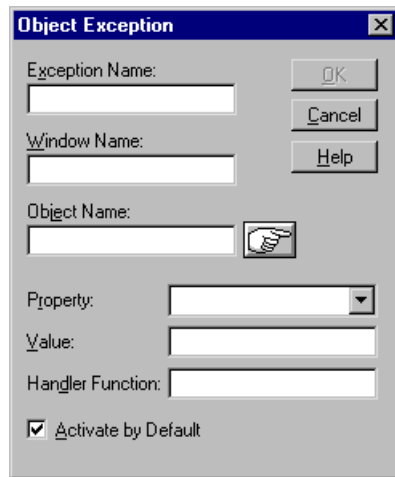
3 In the **Exception Name** box, type in a new name.

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.

- If the object exception you are defining is for a window, enter the name of the window in the **Window Name** box and leave the **Object Name** box empty.

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. For more information, refer to Chapter 5, **Editing the GUI Map**.
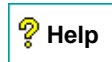
7 In the **Handler Function** box, enter the name of the handler function.

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 554.

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 557. 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*.

Books Online

M Find

Find Again

? Help

Top of Chapter

Back

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, as described in **Defining Object Exceptions** on page 551.

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 23, **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.

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;
}
```

Books Online

M Find

Find Again

? Help

Top of Chapter

⇐ Back

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

Books
Online

Find

Find
Again

Help

Top of
Chapter

Back

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:

- **Statements**
- **Comments and White Space**
- **Constants and Variables**
- **Performing Calculations**
- **Creating Stress Conditions**
- **Decision-Making**
- **Sending Messages to a Report**
- **Starting Applications from a Test Script**
- **Defining Test Steps**
- **Comparing Two Files**

Books Online

Find

Find Again

Help

Top of Chapter

Back

About Enhancing Your Test Scripts with Programming

When you record a test, a test script is generated in Mercury Interactive's Test Script Language (TSL). Each 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 20, **Generating Functions**.

This chapter introduces some basic programming concepts and shows you how to use a few simple programming techniques in order to create more powerful tests. For more information on TSL, refer to the *TSL Online Reference*.

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_check_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.
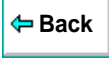
Comments and White Space

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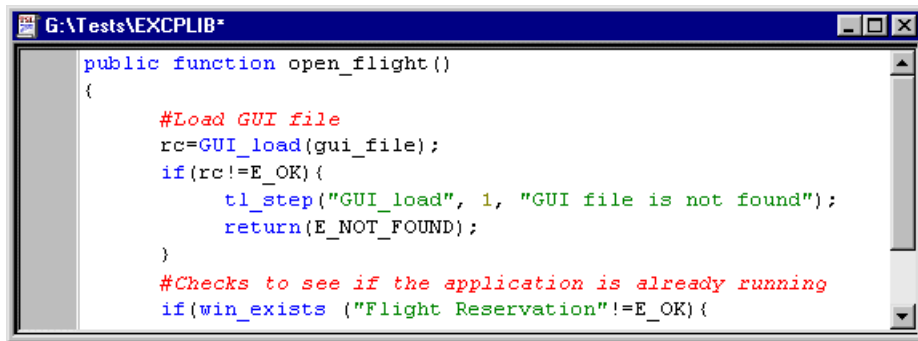
 # Open the Open Order window in Flight Reservation application
set_window ("Flight Reservation", 10);
menu_select_item ("File;Open Order...");

 # 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");

Books Online

Find

Find Again

? Help

Top of Chapter

⇐ Back

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.

```
G:\Tests\EXCPLIB*                                                 _ □ ×
    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 574. For information about variable and constant declarations, see Chapter 22, **Creating User-Defined Functions**.

Performing Calculations

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

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

Books Online

Find

Find Again

Help

Top of Chapter

Back

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

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);
```

Books Online

Find

Find Again

? Help

Top of Chapter

Back

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.

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

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

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

Books Online

M Find

Find Again

? Help

Top of Chapter

Back

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);

Books Online

M Find

Find Again

? Help

Top of Chapter

Back

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 39, **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 folder, then the full path must be given. The optional *save_file* parameter saves the name of a third file, which contains the differences between the first two files.

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

```
# Open win.ini using WordPad.
system("write.exe c:\win95\win.ini");
set_window("win.ini - WordPad",1);

# Save win.ini as win1.ini
menu_select_item("File;Save As...");
set_window("Save As");
edit_set("File Name:_0","c:\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("c:\\win95\\win.ini","c:\\win95\\win1.ini","save");
```

For information on viewing the results of file comparison, see Chapter 28, **Analyzing Test Results**.

Books Online

Find

Find Again

Help

Top of Chapter

Back

Programming with TSL
Generating Functions

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

This chapter describes:

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

About Generating Functions

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

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

- Add Context Sensitive functions that perform operations on a GUI object or get information from the application being tested.

- Add Standard and Analog functions that perform non-Context Sensitive tasks such as synchronizing test execution or sending user-defined messages to a report.

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

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

Books Online

Find

Find Again

? Help

Top of Chapter

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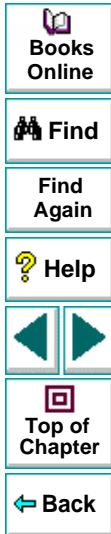
By default, WinRunner suggests the default function for the object. In many cases, this is a **get** function or another function that gets information about the object. For example, if you choose Create > Insert Function > For Object/Window and then click an OK button, WinRunner opens the Function Generator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

button_check_state("OK",1);
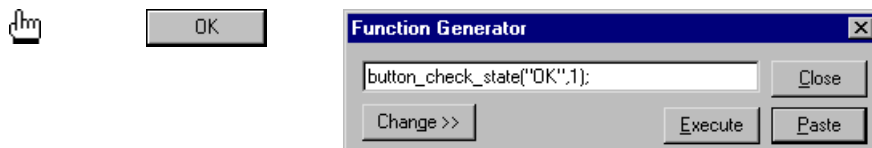
This statement examines the OK button and gets the current value of the enabled property. The *value* can be 1 (enabled), or 0 (disabled).

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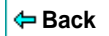
Note that if you point to an object that is not in the GUI map, the object is added automatically to the temporary GUI map file when the generated statement is executed or pasted into the test script.

Note: 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 38, **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 594.

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.

Books Online

M Find

Find Again

? Help

Top of Chapter

Back

To use the default function for a GUI object:

1 Choose **Create > Insert Function > For Object/Window** or click the **Insert Function for Object/Window** button on the User toolbar. WinRunner shrinks to an icon and the mouse pointer becomes a pointing hand.

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

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

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

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

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

Function Generator	☒
button_check_state("OK",1);	Close
Change >>	Execute Paste
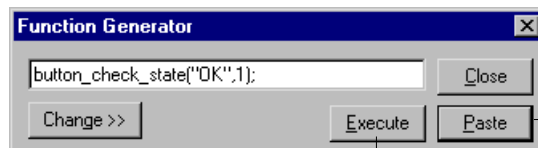
Pastes the function into the script

Executes the function only

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

Selecting a Non-Default Function for a GUI Object

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

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

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

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

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

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

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

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

Function Generator

```
button_wait_info("OK","enabled",1,10);                Close
Category:        push_button                    ▼    Paste
Function Name:   button_wait_info                     Execute
                 button_check_info          ▲
                 button_get_info                      Help
                 button_press
                 button_wait_info           ▼
Description:
This function waits for a specified value of a button attribute  ▲
arg1 is the button name
arg2 is the requested attribute
arg3 is the expected value                                        ▼
Set as Default                                       Args >>
```
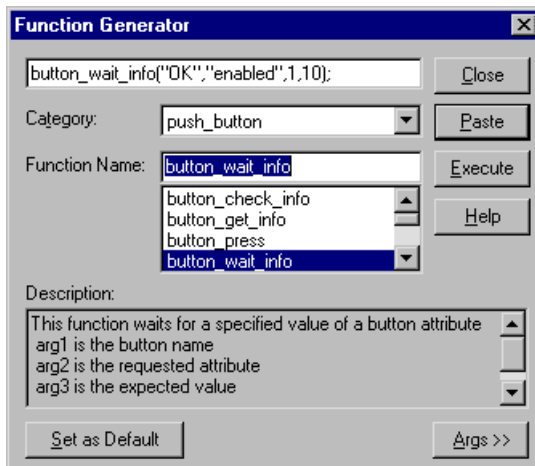
6 If you want to modify the argument values, click **Args**. The dialog box expands and displays a text box for each argument. See **Assigning Argument Values** on page 591 to learn how to fill in the argument text boxes.

 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.

Books Online

Find

Find Again

Help

Top of Chapter

Back

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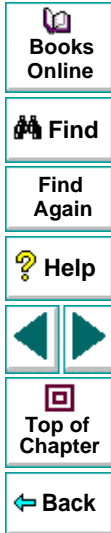
When programming a test, perhaps you know the task you want the test to perform but not the exact function to use. The Function Generator helps you to quickly locate the function you need and insert it into your test script. Functions are organized by category; you select the appropriate category and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

Books Online

Find

Find Again

? Help

Top of Chapter

← Back

To select a function from a list:

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

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

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

4 To define or modify the argument values, click **Args**. The dialog box expands and displays a text box for each argument. See **Assigning Argument Values** on page 591 to learn how to fill in the argument text boxes.

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

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

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

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

Assigning Argument Values

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

To assign or modify argument values for a generated function:

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

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

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

4 Click **Args**. The dialog box expands based on the number of arguments in the function.
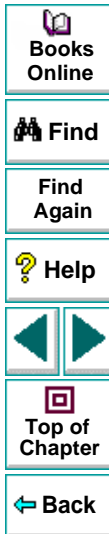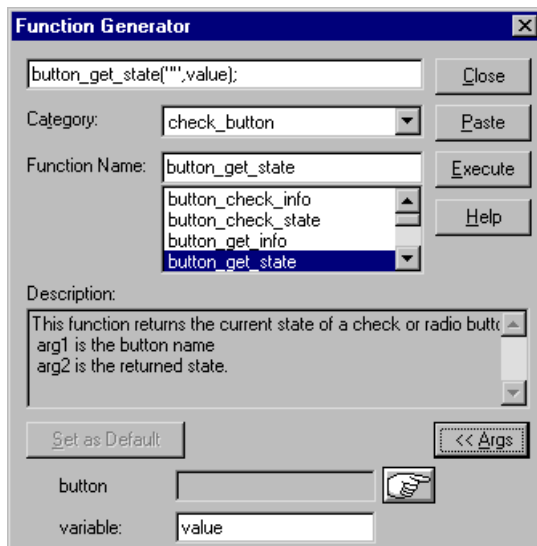
5 Assign values to the arguments. You can assign a value either manually or automatically.

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

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

Note that if you click an object that is not compatible with the selected function, a message 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_text**. For Analog, Standard and Customization function categories, the default is the most commonly used function in the category. For example, the default function for the system category is **invoke_application**.

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

To change the default function in a category:

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

2 In the **Category** list, select a function category. For example, if you want to view menu functions, select menu.

3 In the **Function Name** list, select the function that you want to make the default.

4 Click **Set as Default**.

5 Click **Close**.
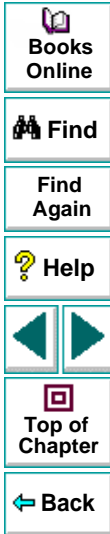
The selected function remains the default function in its category until it is changed or until you end your WinRunner session. To save changes to the default function setting, add a **generator_set_default_function** statement to your startup test. For more information on startup tests, see Chapter 39, **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.

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:

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

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 29, **Running Batch Tests**.

Using the Call Statement

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

- A **call** statement invokes a test from within another test.

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

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

call *test_name* **([** *parameter₁*, *parameter₂*, ...*parameterₙ* **]);**

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

call_close *test_name* **([** *parameter₁*, parameter₂, ... parameterₙ **]);**

The *test_name* is the name of the test to invoke. The *parameters* are the parameters defined for the called test.

The 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 folder specified in the search path, or else be called with the full pathname enclosed within quotation marks.

For example:

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

While running a called test, you can pause execution and view the current call chain. To do so, choose Debug > Calls.
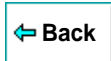
Returning to the Calling Test

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.

Books Online

Find

Find Again

Help

Top of Chapter

Back

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 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 29, **Running Batch Tests**.

Books Online

Find

Find Again

Help

Top of Chapter

Back

Setting the Search Path

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

To set the search path, choose Settings > General Options. The General 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 folder 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.

For more information about how to set a search path in the General Options dialog box, see Chapter 36, **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.

General Options ⊠

| Record | Run | Miscellaneous |
| Text Recognition | Environment | Folders | Current Test |

Please specify folders for:

Temporary files: `D:\Program Files\Mercury Interactive\Wi` [...]

Temporary GUI map file: `D:\Program Files\Mercury Interactive\Wi` [...]

Shared checklists: `D:\Program Files\Mercury Interactive\Wi` [...]

Documentation files: [...]

Search path for called tests:

[_____] [...]   [Add]

`D:\Program Files\Mercury Interactive\WinRunner\samples`   [Delete]

`.`

`D:\Program Files\Mercury Interactive\WinRunner\lib`   [Up]

[Down]

[OK] [Cancel] [Apply] [Help]

Books Online

Find

Find Again

Help

Top of Chapter

Back

For example:

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

This statement tells WinRunner to search the *c:\ui_tests* folder for called tests. For more information on using the **setvar** function, see Chapter 37, **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 1014.

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 **File** > **Test Properties**.

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.

Books Online

Find

Find Again

Help

Top of Chapter

Back

 # test_a, with parameter k

 # Note that the ampersand (&) is a bitwise AND operator. It signifies concatenation.

```
i = 1;
j = 2;
k = 3;
call test_b(i);
print(j & k & l); # prints '256'
```

 # test_b, with parameter j

 # Note that the ampersand (&) is a bitwise AND operator. It signifies concatenation.

```
j = 4;
k = 5;
l = 6;
print(i & j & k); # prints '145'
```

Programming with TSL
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:

- **Function Syntax**

- **Return Statements**

- **Variable, Constant, and Array Declarations**

- **Example of a User-Defined Function**

About Creating User-Defined Functions

In addition to providing built-in functions, TSL allows you to design and implement your own functions. You can:

- Create user-defined functions in a test script. You define the function once, and then you 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 23, **Creating Compiled Modules**.

- Call functions from the Microsoft Windows API or any other external functions stored in a DLL. For more information, see Chapter 24, **Calling Functions from External Libraries**.

Books Online

Find

Find Again

? Help

Top of Chapter

Back

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.

Books Online

M Find

Find Again

? Help

Top of Chapter

Back

Function Syntax

A user-defined function has the following structure:

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

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.

Books Online

Find

Find Again

Help

Top of Chapter

Back

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 Statements

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

return [(*expression*)]**;**

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

Variable, Constant, and Array Declarations

Declaration is usually optional in TSL. In functions, however, variables, constants, and arrays must all be declared. The declaration can be within the function itself, or anywhere else within the test script or compiled module containing the function. You can find additional information about declarations in the *TSL 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. This variable is initialized each time the definition of the function is executed.

Note: In compiled modules, a **static** variable is initialized whenever the compiled module is compiled.

---

public: A public variable can be declared only within a test or module, and is available for all functions, tests, and compiled modules.

extern: An extern declaration indicates a reference to a public variable declared outside of the current test or module.

Remember that you must declare all variables used in a function within the function itself, or within the test or module that contains the function. If you wish to use a public variable that is declared outside of the relevant test or module, you must declare it again as **extern**.

The **extern** declaration must appear within a test or module, before the function code. An extern declaration cannot initialize a variable.

For example, suppose that in Test 1 you declare a variable as follows:

public window_color=green;

In Test 2, you write a user-defined function that accesses the variable window_color. Within the test or module containing the function, you declare window_color as follows:

extern window_color;

With the exception of the **auto** variable, all variables continue to exist until the Stop command is executed.

Note: In compiled modules, all variables continue to exist until the Stop command is executed with the exception of the **auto** and **public** variables. (The **auto** variables exist only as long as the function is running; **public** variables exist until exiting WinRunner.)

The following table summarizes the scope, lifetime, and availability (where the declaration can appear) of each type of variable:

Declaration	Scope	Lifetime	Declare the Variable in...
auto	local	end of function	function
static	local	until abort	function, test, or module
public	global	until abort	test or module
extern	global	until abort	function, test, or module

Note: In compiled modules, the Stop command initializes **static** and **public** variables. For more information, see Chapter 23, **Creating Compiled Modules**.

Constants

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

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

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

    #Compare label to file name
    if ( file != lbl)
        return 1;
    else
        return 0;
}
rc=open_file("c:\\dash\\readme.tx");
pause(rc);
```

Programming with TSL
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:

- **Contents of a Compiled Module**
- **Creating a Compiled Module**
- **Loading and Unloading a Compiled Module**
- **Example of a Compiled Module**

About Creating 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.

Books Online

Find

Find Again

Help

Top of Chapter

Back

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 39, **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. For more information, see **Creating a Compiled Module** on page 630.

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 22, **Creating User-Defined Functions**.

- prototypes of external functions. For more information, see Chapter 24, **Calling Functions from External Libraries**.

- **load** statements to other modules. For more information, see **Loading and Unloading a Compiled Module** on page 632.

Books Online

Find

Find Again

Help

Top of Chapter

Back

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.

Books Online

Find

Find Again

Help

Top of Chapter

Back

Creating a Compiled Module

Creating a compiled module is similar to creating a regular test script.

To create a compiled module:

1 Choose **File > Open** to open a new test.

2 Write the user-defined functions.

3 Choose **File > Test Properties**. The **Test Properties** dialog box opens.

Books Online

Find

Find Again

Help

Top of Chapter

Back

4 In the **Test Type** box, click **Compiled Module** and then click **OK**.
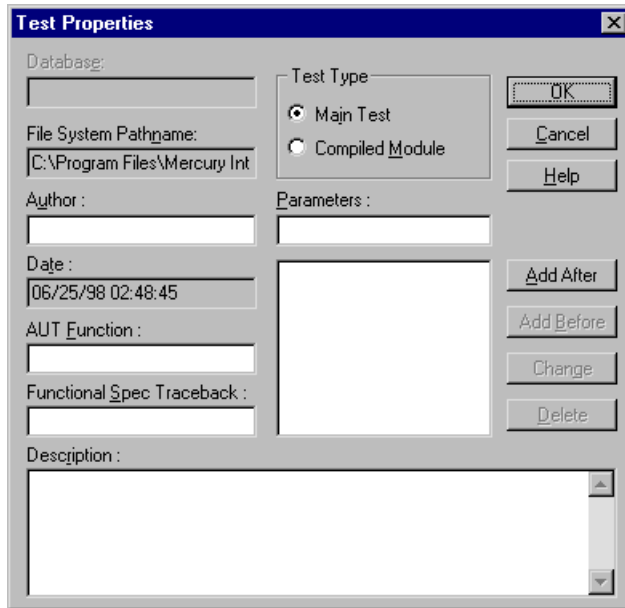
5 Choose **File > Save**.

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 **Setting the Search Path** on page 603.

6 Compile the module using the **load** function. For more information, see **Loading and Unloading a Compiled Module** below.

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 39, **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.

Note: If you make changes to a function in a loaded compiled module, you must unload and reload the compiled module in order for the changes to take effect.

---

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)

Books Online

Find

Find Again

Help

Top of Chapter

Back

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 module; it decrements the load counter. When execution returns to test A, *math_functions* is still loaded.

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;
}
```

Programming with TSL
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:

- **Dynamically Loading External Libraries**
- **Declaring External Functions in TSL**
- **Windows API Examples**

Books Online

Find

Find Again

? Help

Top of Chapter

Back

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.

Books Online

Find

Find Again

? Help

Top of Chapter

← Back

You can call any function exported from a DLL with the _ _ stdcall calling convention. You can also load DLLs that are part of the application being tested in order to access its exported functions.

Using the **load_dll** function, you dynamically load the libraries containing the functions you need. Before you actually call the function, you must write an *extern* declaration so that the interpreter knows that the function resides in an external library.

Note: 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 20, **Generating Functions**. 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** folder in the installation folder.

**Books
Online**

M Find

**Find
Again**

? Help

**Top of
Chapter**

⇦ Back

Dynamically Loading External Libraries

In order to load the external DLLs (Dynamic Link Libraries) containing the functions you want to call, use the TSL function **load_dll**. This function performs a runtime load of a 32-bit DLL. It has the following syntax:

load_dll (*pathname* **);**

The *pathname* is the full pathname of the DLL to be loaded.

For example:

load_dll ("h:\\qa_libs\\os_check.dll");

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.

To unload a loaded external DLL, use the TSL function **unload_dll**. It has the following syntax:

unload_dll (*pathname* **);**

For example:

unload_dll ("h:\\qa_libs\\os_check.dll");

The *pathname* is the full pathname of the 32-bit DLL to be unloaded.

To unload all loaded 32-bit DLLs from memory, use the following statement:

unload_dll ("");

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 16-bit 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 39, **Initializing Special Configurations**.)

The syntax of the extern declaration is:

extern *type function_name* **(***parameter1*, *parameter2*,...**);**

The *type* refers to the return value of the function. The type can be one of the following:

char (signed and unsigned)	*float*
short (signed and unsigned)	*double*
int (signed and unsigned)	*string* (equivalent to C char*)

Each *parameter* must include the following information:

[mode]   *type*   [name]   [*<size>*]

The *mode* can be either *in*, *out*, or *inout*. The default is *in*. Note that these values must appear in lowercase letters.

The *type* can be any of the values listed above.

An optional *name* can be assigned to the parameter to improve readability.

The *<size>* is required only for an *out* or *inout* parameter of type *string* (see below.)

For example, suppose you want to call a function called set_clock that sets the time on a clock application. The function is part of an external DLL that you loaded with the **load_dll** statement. To declare the function, write:

extern int set_clock (string name, int time);

The set_clock function accepts two parameters. Since they are both input parameters, no mode is specified. The first parameter, a string, is the name of the clock window. The second parameter specifies the time to be set on the clock. The function returns an integer that indicates whether the operation succeeded.

Once the extern declaration is interpreted, you can call the set_clock function the same way you call a TSL function:

result = set_clock ("clock v. 3.0", 3);

Books Online

Find

Find Again

Help

Top of Chapter

Back

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 external function must adhere to the following conventions:

● Any parameter designated as a *string* in TSL must correspond to a parameter of type *char**.

● Any parameter of mode *out* or *inout* in TSL must correspond to a pointer in your exported function. For instance, a parameter *out int* in TSL must correspond to a parameter *int** in the exported function.

● The external function must observe the standard Pascal calling convention *export far Pascal*.

For example, the following declaration in TSL:

extern int set_clock (string name, inout int time);

must appear as follows in your external function:

```
int set_clock(
      char* name,
      int* time
      );
```

Windows API Examples

The following sample tests call functions in the Windows API.

Checking Window Mnemonics

This test integrates the API function *GetWindowTextA* into a TSL function that checks for mnemonics (underlined letters used for keyboard shortcuts) in object labels. The TSL function receives one parameter: the logical name of an object. If a mnemonic is not found in an object's label, a message is sent to a report.
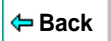
```
# load the appropriate DLL (from Windows folder)
load ("win32api");

# define the user-defined function "check_labels"
public function check_labels(in obj)
{
    auto hWnd,title,pos,win;
    win = GUI_get_window();
    obj_get_info(obj,"handle",hWnd);
    GetWindowTextA(hWnd,title,128);
    pos = index(title,"&");
    if (pos == 0)
        report_msg("No mnemonic for object: "& obj & "in window: "& win);
}

# start Notepad application
invoke_application("notepad.exe","","",SW_SHOW);

# open Find window
set_window ("Notepad");
menu_select_item ("Search;Find...");

# check mnemonics in "Up" radio button and "Cancel" pushbutton
set_window ("Find");
check_labels ("Up");
check_labels ("Cancel");
```

Books
Online

M Find

Find
Again

? Help

回
Top of
Chapter

← Back

Loading a DLL and External Function

This test fragment uses crk_w.dll to prevent recording on a debugging application. To do so, it calls the external *set_debugger_pid* function.

```
# load the appropriate DLL
load_dll("crk_w.dll");

# declare function
extern int set_debugger_pid(long);

# load Systems DLLs (from Windows folder)
load ("win32api");

# find debugger process ID
win_get_info("Debugger","handle",hwnd);
GetWindowThreadProcessId(hwnd,Proc);

# notify WinRunner of the debugger process ID
set_debugger_pid(Proc);
```

Programming with TSL
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:

- **Creating an Input Dialog Box**
- **Creating a List Dialog Box**
- **Creating a Custom Dialog Box**
- **Creating a Browse Dialog Box**
- **Creating a Password Dialog Box**

Books Online

Find

Find Again

? Help

Top of Chapter

⬅ Back

About Creating Dialog Boxes for Interactive 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.

Books Online

Find

Find Again

? Help

Top of Chapter

Back

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.
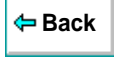
Creating an Input Dialog Box

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.

Books Online

M Find

Find Again

? Help

Top of Chapter

⬅ Back

To create such a dialog box, you would program the following statement:

name = create_input_dialog ("Please type in your name.");
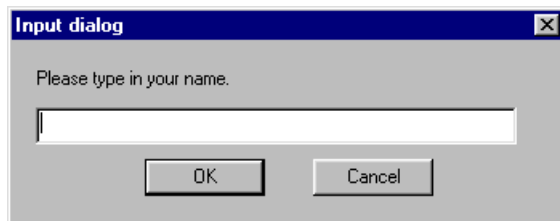
The input that is typed into the dialog box during test execution is passed to the variable *name* when the OK button is clicked. If the Cancel button is clicked, an empty string (empty quotation marks) is passed to the variable *name*.

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.
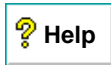
Creating a List Dialog Box

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");
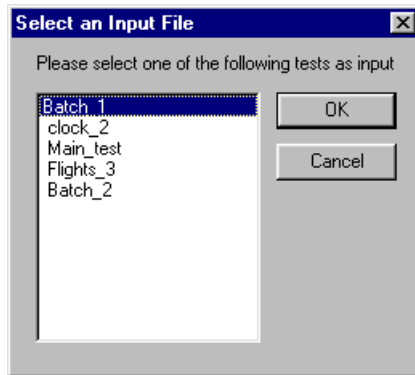
The item that is selected from the list during test execution is passed to the variable *filename* when the OK button is clicked. If the Cancel button is clicked, an empty string (empty quotation marks) is passed to the variable *filename*.

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$_{1-n}$, check_name$_{1-m}$* **);**

- *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).

Books Online

M Find

Find Again

? Help

Top of Chapter

Back

When the "execute" button is clicked, the values that the user enters are passed as parameters to the specified function, in the following order:

$edit\_name_1,...\ edit\_name_n\ ,check\_name_1,...\ check\_name_m$

In the following example, the custom dialog box allows the user to specify startup parameters for an application. When the user clicks the Run button, the user-defined function, run_application1, invokes the specified Windows application with the initial conditions that the user supplied.

res = create_custom_dialog ("run_application1", "Initial Conditions", "Run",
    "Application:, Geometry:, Background:, Foreground:, Font:", "Sound,
    Speed");
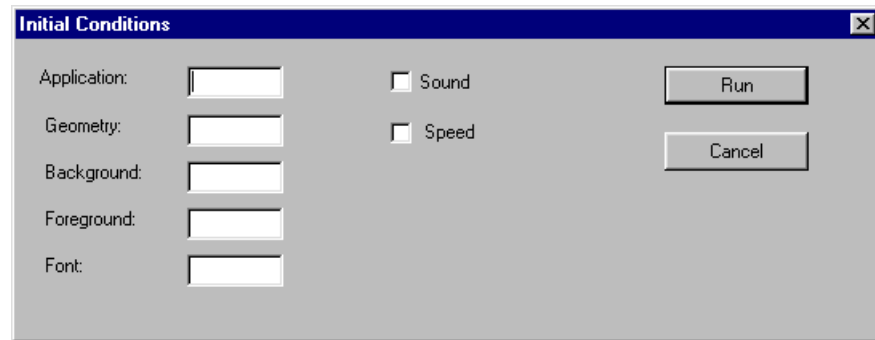
If the specified function returns a value, this value is passed to the variable *res*. If the Cancel button is clicked, an empty string (empty quotation marks) is passed to the variable *res*.

Note that you can precede any edit field label with an exclamation mark. When the user types input into the edit field, each character entered is represented by an asterisk. You use an exclamation mark to prevent others from seeing confidential information, such as a password.

Creating a Browse Dialog Box

A browse dialog box allows you to select a file from a list of files, and returns the name of the selected file as a string.

You use the TSL function **create_browse_file_dialog** to create a browse dialog box. This function has the following syntax:

create_browse_file_dialog (*filter* **);**

where *filter* sets a filter for the files to display in the Browse dialog box. You can use wildcards to display all files (*.*) or only selected files (*.exe or *.txt etc.).
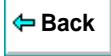
In the following example, the browse dialog box displays all files with extensions .dll or .exe.

filename = create_browse_file_dialog("*.dll;*.exe");
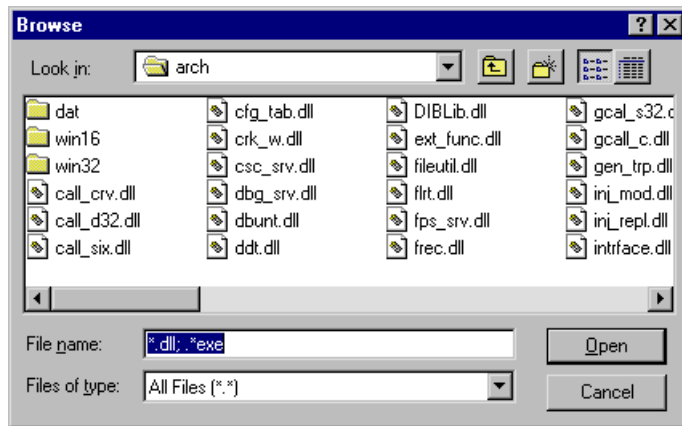
When the OK button is clicked, the name and path of the selected file is passed to the variable *filename*. If the Cancel button is clicked, an empty string (empty quotation marks) is passed to the variable *filename*.
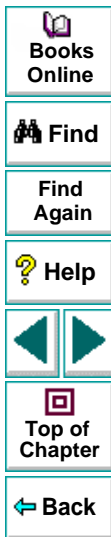
Creating a Password Dialog Box

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 the interactive test run is saved to variables for analysis.

You use the TSL function **create_password_dialog** to create a password dialog box. This function has the following syntax:

create_password_dialog (*login, password, login_out, password_out* **);**

- *login* is the label of the first edit field, used for user-name input. If you specify an empty string (empty quotation marks), the default label "Login" is displayed.

- *password* is the label of the second edit field, used for password input. If you specify an empty string (empty quotation marks), the default label "Password" is displayed. When the user enters input into this edit field, the characters do not appear on the screen, but are represented by asterisks.

- *login_out* is the name 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);
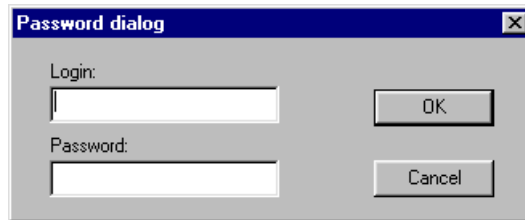
If the OK button is clicked, the value 1 is passed to the variable *status*. If the Cancel button is clicked, the value 0 is passed to the variable *status* and the *login_out* and *password_out* parameters are assigned empty strings.

Programming with TSL
Using Regular Expressions

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

A regular expression is a string that specifies a complex search phrase. In most cases the string is preceded by an exclamation point (!). (In descriptions or arguments of functions where a string is expected, such as the **match** function, the exclamation point is not required.) By using special characters such as a period (.), asterisk (*), caret (^), and brackets ([ ]), you define the conditions of the search. For example, the string "!windo.*" matches both "window" and "windows". See **Regular Expression Syntax** on page 669 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 following instances:

- the physical description of an object in the GUI map

- a GUI checkpoint, when evaluating the contents of an edit object or static text object with a varying name

- a text checkpoint, to locate a varying text string using **win_find_text** or **object_find_text**

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

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

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

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

Books Online

Find

Find Again

Help

Top of Chapter

Back

Using a Regular Expression in a GUI Checkpoint

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

To define a regular expression in a GUI checkpoint:

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

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

2 Double-click a static text object.

Books Online

Find

Find Again

Help

Top of Chapter

Back

3 The **Check GUI** dialog box opens:
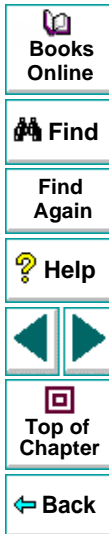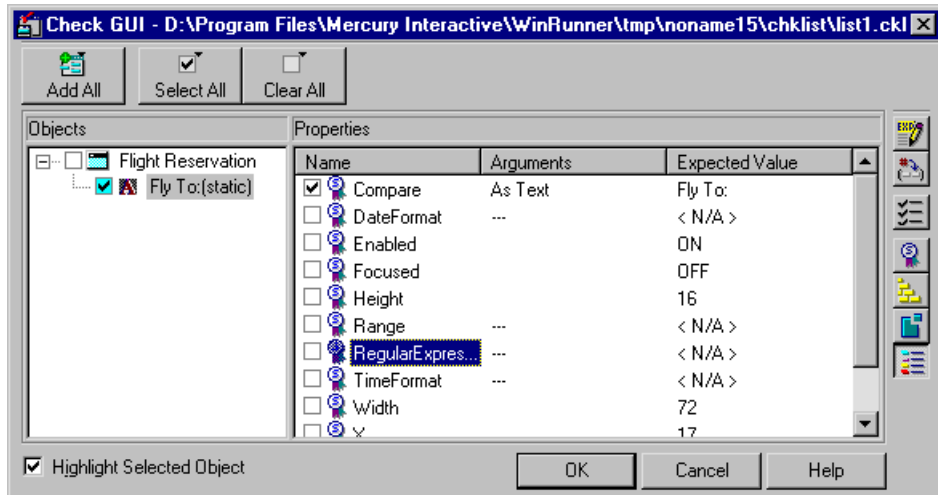
4 In the **Properties** pane, highlight the "Regular Expression" property check and then click the **Specify Arguments** button.

The **Check Arguments** dialog box opens:

Check Arguments	✕
┌─ Please enter regular expression: ─────	
Regular Expression: [ ]	
[OK] [Cancel] [Help]	
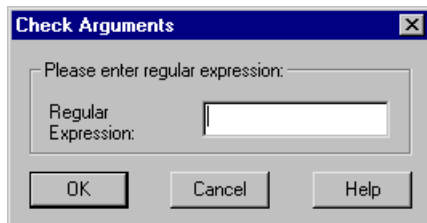
5 Enter the regular expression in the **Regular Expression** box, and then click **OK**.

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

---

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

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

For additional information on specifying arguments, see **Specifying Arguments for Property Checks** on page 252.

Using a Regular Expression in a Text Checkpoint

You can use a regular expression in a text checkpoint, to locate a varying text string using **win_find_text** or **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**.

Books Online

Find

Find Again

Help

Top of Chapter

Back

Regular Expression Syntax

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

The 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 (^).

**Books
Online**

Find

**Find
Again**

Help

**Top of
Chapter**

Back

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

[^A-Za-z]

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

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

[]g-m]

matches the "]" and g through m.

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

Matching Specific Characters

An asterisk (*) instructs WinRunner to match 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]*"
}
```
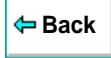
Running Tests

Once you have developed a test script, you run the test to check the behavior of your application.

This chapter describes:

- **WinRunner Test Run Modes**
- **WinRunner Run Commands**
- **Choosing Run Commands Using Softkeys**
- **Running a Test to Check Your Application**
- **Running a Test to Debug Your Test Script**
- **Running a Test to Update Expected Results**
- **Controlling Test Execution with Testing Options**

Books Online

Find

Find Again

Help

Top of Chapter

Back

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 on the Standard toolbar. The Verify mode is the default run mode.

Use WinRunner's Run menu 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 29, **Running Batch Tests**.

Books Online

M Find

Find Again

? Help

Top of Chapter

⇐ Back
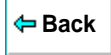
WinRunner Test Run Modes

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 28, **Analyzing Test Results**.

You can save as many sets of verification results as you need. To do so, save the results in a new folder each time you run the test. You specify the folder 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 686.

Books Online

M Find

Find Again

? Help

Top of Chapter

⇐ Back

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* folder. 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 Tools > Test Results.

Once you run a test in Debug mode, that 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 31, **Debugging Test Scripts**.

● Set breakpoints to pause execution at specified points in the test script. For more information, see Chapter 32, **Using Breakpoints**.

● Use the Watch List to monitor variables in a test script during execution. For more information, see Chapter 33, **Monitoring Variables**.

For more information about running a test script in Debug mode, see **Running a Test to Debug Your Test Script** on page 688.

Update

Use the Update mode to update the *expected* results of a test or to create a new expected results folder. 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 691.

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* folder, 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 689.

Books Online
Find
Find Again
Help
Top of Chapter
Back

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

Choose the **Run from Top** command or click the corresponding **Run from Top** button to run 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

Choose the **Run from Arrow** command or click the corresponding **Run from Arrow** button to run 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. The following Run Minimized commands are available:

- **Run Minimized > From Top** command

- **Run Minimized > From Arrow** command

Step Commands

You use a Step command or click a Step button to run a single statement in a test script. For more information on the Step commands, see Chapter 31, **Debugging Test Scripts**. The following Step commands are available:

- **Step** command
- **Step Into** command
- **Step Out** command
- **Step to Cursor** command

    The following Step buttons are available:

Step button

Step Into button

Stop

You can stop a test run immediately by choosing the **Stop** command or clicking the **Stop** button. 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 695 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 23, **Creating Compiled Modules**.

Pause

You can pause a test by choosing the **Pause** command or clicking the **Pause** button. 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 695 for more information.

To resume running a paused test, choose the appropriate Run command. Test execution resumes from the point where 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 34, **Customizing WinRunner's User Interface**.

The following table lists the default softkey configurations for running tests:

Command	Default Softkey Combination	Function
RUN FROM TOP	Ctrl Left + F5	Runs the test from the beginning.
RUN FROM ARROW	Ctrl Left + F7	Runs the test from the line in the script indicated by the arrow.
STEP	F6	Runs only the current line of the test script.
STEP INTO	Ctrl Left + F8	Like Step: however, if the current line calls a test or function, the called test or function 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 the test run after all previously interpreted TSL statements have been executed. Execution can be resumed from this point.
STOP	Ctrl Left + F3	Stops the test run.

Books Online

Find

Find Again

Help

Top of Chapter

Back
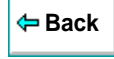
Running a Test to Check Your Application

When you run a test to check the behavior of your application, WinRunner compares the current results with the expected results. You specify the folder in which to save the verification results for the test.

To run a test to check your application:

1 If your test is not already open, choose **File > Open** or click the **Open** button to open the test.

2 Make sure that **Verify** is selected from the drop-down list of run modes on the Standard toolbar.

3 Choose the appropriate **Run** menu command or click one of the **Run** buttons.

The **Run Test** dialog box opens, displaying a default test run name for the verification results.

Run Test	⊠
Test Run Name: res1 ▼	OK
	Cancel
☐ Use Debug mode (don't display this dialog box)	Help
☑ Display test results at end of run	
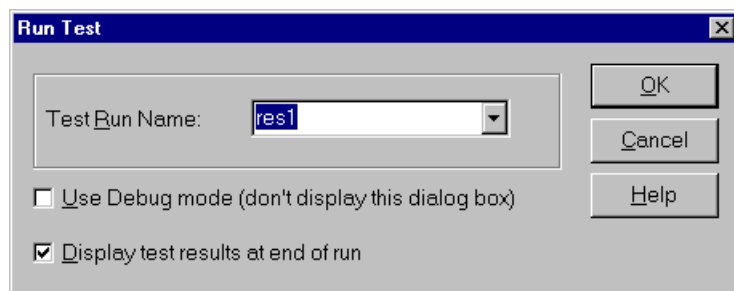
4 You can save the test results under the default test run name. To use a different name, type in a new name or select an existing name from the list.

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 with the test run name you specified.

Books Online

M Find

Find Again

? Help

Top of Chapter

⇐ Back

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 folder. 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 If your test is not already open, choose **File > Open** to open the test.

2 Select **Debug** from the drop-down list of run modes on the Standard toolbar.

3 Choose the appropriate **Run** menu command.

To execute the entire test, choose **Run > Run from Top** or click the **Run from Top** button. The test runs from the top of the test script and generates a set of debug results.

To run part of the test, choose one of the following commands or click one of the corresponding buttons:

Run > Run from Arrow

Run > Run Minimized > From Arrow

Run > Step

Run > Step Into

Run > Step Out

Run > Step to Cursor

The test runs according to the command you chose, and generates a set of debug results.

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 If your test is not already open, choose **File > Open** to open the test.

2 Select **Update** from the list of run modes on the Standard toolbar.

3 Choose the appropriate **Run** menu command.

To update the entire set of expected results, choose **Run > Run from Top** or click the **Run from Top** button.

To update only a portion of the expected results, choose one of the following commands or click one of the corresponding buttons:

Run > Run from Arrow

Run > Run Minimized > From Arrow

Run > Step

Run > Step Into

Run > Step Out

Run > Step to Cursor

WinRunner runs the test according to the Run menu command you chose and updates the expected results. The default folder for expected results is *exp*.
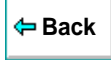
Generating Multiple Expected Results

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 **File > Open** or click the **Open** button. The Open Test dialog box opens.

Books Online

M Find

Find Again

? Help

Top of Chapter

Back
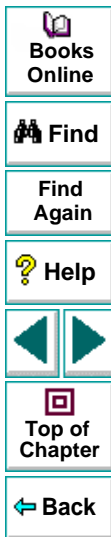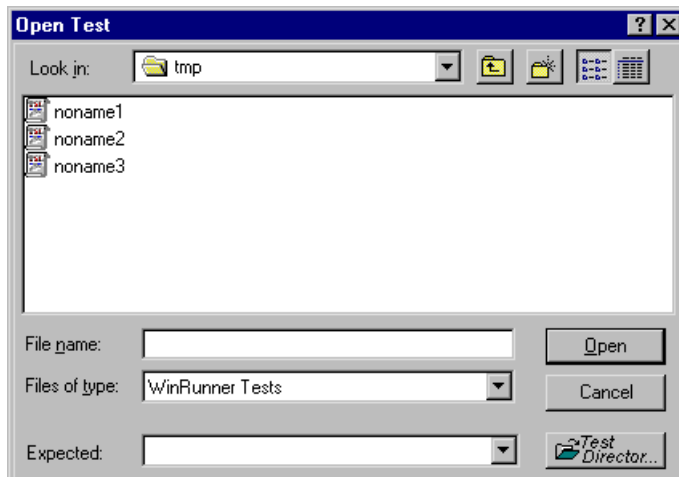
2 In the **Open Test** dialog box, select the test for which you want to create multiple sets of expected results. In the **Expected** box, type in a unique folder name for the new expected results.

Note: To create a new set of expected results for a test that is already open, choose **File > Open** or click the **Open** button to open the Open Test dialog box, select the open test, and then enter a name for a new expected results folder in the **Expected** box.

---

3 Click **OK**. The Open Test dialog box closes.

4 Choose **Update** from the list of run modes on the Standard toolbar.

5 Choose **Run > Run from Top** or click the **Run from Top** button to generate a new set of expected results.

WinRunner runs the test and generates a new set of expected results, in the folder 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 **File > Open** or click the **Open** button. The Open Test dialog box opens.

Note: If the test is already open, but it is accessing the wrong set of expected results, you must choose **File > Open** or click the **Open** button to open the Open Test dialog box again, next select the open test, and then choose the correct expected results folder 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 **Open**. The Open Test dialog box closes.

 4 Select **Verify** from the drop-down list of run modes on the Standard toolbar.

Books Online

Find

Find Again

Help

Top of Chapter

Back

5 Choose the appropriate **Run** menu command. The **Run Test** dialog box opens, displaying a default test run name for the verification results—for example, *res1*.
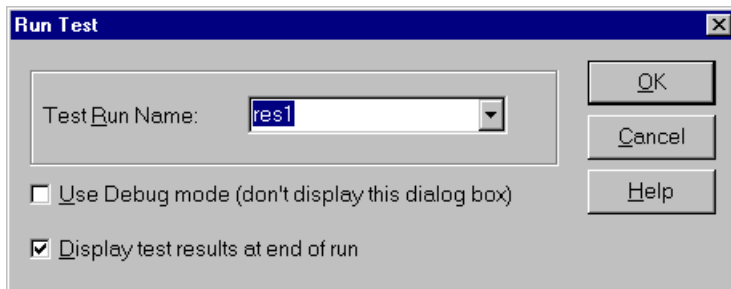
6 Click **OK** to begin test execution, and to save the test results in the default folder. To use a different verification results folder, type in a new name or choose an existing name from the list.

The Run Test dialog box closes. WinRunner runs the test according to the Run menu command you chose and saves the test results in the folder you specified.
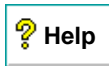
Controlling Test Execution with Testing Options

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 General Options dialog box. Choose **Settings > General Options** 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 General Options dialog box. For a more comprehensive discussion of setting testing options globally, see Chapter 36, **Setting Global Testing Options**.

You can also set the corresponding *min_diff* option from within a test script using the **setvar** function. For a more comprehensive discussion of setting testing options from within a test script, see Chapter 37, **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.

You can view the settings for the current test in the Current Test tab of the General Options dialog box. For more information about viewing the settings for the current test, see Chapter 36, **Setting Global Testing Options**.

Running Tests
Analyzing Test Results

After you run a test, you can view a report of all the major events that occurred during the test run.

This chapter describes:

- **The Test Results Window**
- **Viewing the Results of a Test Run**
- **Viewing the Results of a GUI Checkpoint**
- **Viewing the Results of a GUI Checkpoint on Table Contents**
- **Viewing the Expected Results of a GUI Checkpoint on Table Contents**
- **Viewing the Results of a Bitmap Checkpoint**
- **Viewing the Results of a Database Checkpoint**
- **Viewing the Expected Results of a Content Check in a Database Checkpoint**
- **Updating the Expected Results of a Checkpoint**
- **Viewing the Results of a File Comparison**
- **Reporting Defects Detected during a Test Run**

Books Online

Find

Find Again

Help

Top of Chapter

Back

About Analyzing Test Results

After you run a test, test results are displayed in the Test Results window. This window contains a description of the major events that occurred during the test run, such as GUI, bitmap, or database checkpoints, file comparisons, and error messages. It also includes tables and pictures to help you analyze the results.

The Test Results Window

After a test run, you can view test results in the Test Results window. To open the window, choose Tools > Test Results or click the Test Results button.

Results

The results box enables you to choose which results to display for the test. You can view the expected results (*exp*) or the actual results for a specified test run.

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.

Books Online

Find

Find Again

Help

Top of Chapter

Back
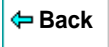
Test Summary

The following information appears in the test summary:

Test Results

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 test1 (5)

indicates the first bitmap checkpoint, in a test called *test1*, 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. Double-click the detailed description of the bitmap checkpoint to display the contents of the bitmap checkpoint, as shown in the previous example. For more information, see **Viewing the Results of a Bitmap Checkpoint** on page 725.

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 test1 (10)

indicates the first GUI checkpoint in a test called *test1*, in the tenth line of the test script. The number in parentheses indicates the line in the test script that contains the **obj_check_gui** or **win_check_gui** statement. Double-click the detailed description of the GUI checkpoint to display the GUI Checkpoint Results dialog box for that checkpoint. For more information, see **Viewing the Results of a GUI Checkpoint** on page 710.

General Information

Double-click the General Information icon 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 Folder: The name of the expected results folder 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.

Books Online

Find

Find Again

Help

Top of Chapter

Back

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.

To view the results of a non-active test, choose **File > Open**. In the **Open Test Results** dialog box, select the test whose results you want to view.

Note that if you ran a test in Verify mode and the **Display Test Results at End of Run** check box 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 27, **Running Tests**.

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** 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** or click the **Test Results** button in the main WinRunner window.

The **Test Results** window opens, displaying the test results of the latest 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.
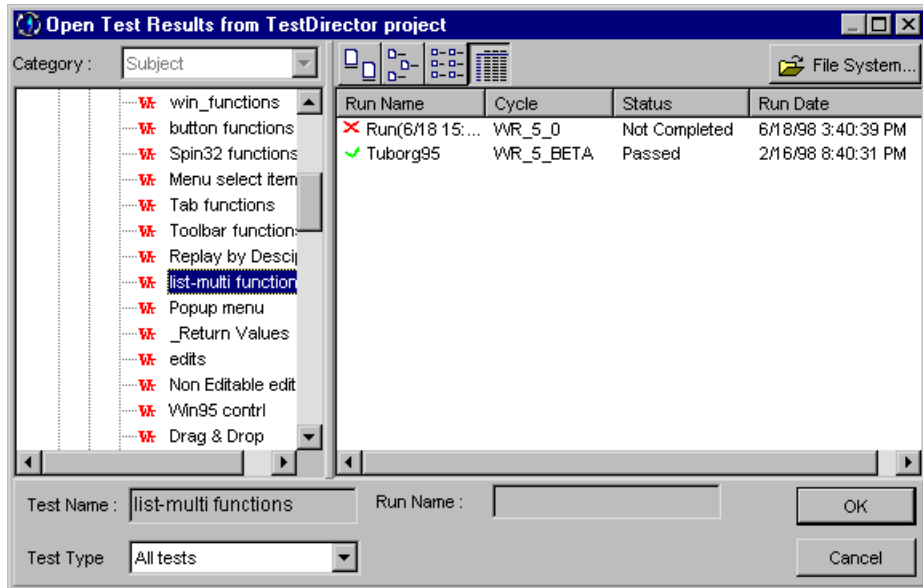
3 In the **Test Type** box, select the type of test to view in the dialog box: **WinRunner Tests** (the default setting), **WinRunner Batch Tests**, or **All Tests**.

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 **Test Set** column contains the names of the test sets. Entries in the **Status** column indicate whether the test passed or failed. The **Run Date** column displays the date and time when the test set was run.

6 Click **OK** to view the results of the selected test.

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 40, **Managing the Testing Process**.

Viewing the Results of a Property Check

A property check helps you to identify specific changes in the behavior of GUI objects in your application. For example, you can check whether a button is enabled or disabled or whether an item in a list is selected. The results of a property check are displayed in the Property dialog box that you open from the Test Results window. The expected and actual results are shown.

For more information, see Chapter 9, **Checking GUI Objects**.

To display the results of a property check:

1 Choose **Tools > Test Results** or click the **Test Results** button in the main WinRunner window to open the **Test Results** window.

2 In the test log, look for entries that list "property check" in the **Event** column. Failed checks appear in red; passed checks appear in green.

3 Double-click an "property check" entry in the test log. The **Property** dialog box opens. It displays the expected and actual values.

Property "enabled" for push_button "OK"	☒
Expected value:	Actual value:
1	0
OK	
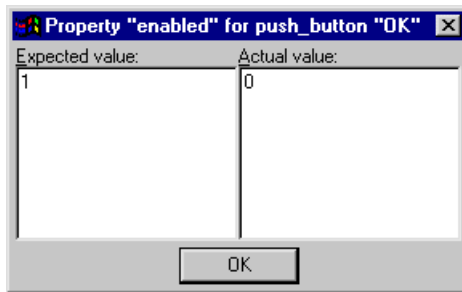
4 Click **OK** to close the dialog box.
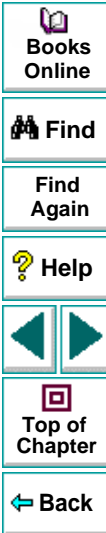
Viewing the Results of a GUI Checkpoint

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 displayed 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 Choose **Tools > Test Results** or click the **Test Results** button in the main WinRunner window to open the **Test Results** window.

 2 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.

 3 Double-click an "end GUI checkpoint" entry in the test log. Alternatively, highlight the entry and choose **Options > Display** or click the **Display** button. The **GUI Checkpoint Results** dialog box opens.

Books Online

M Find

Find Again

? Help

Top of Chapter

← Back

The GUI Checkpoint Results dialog box lists the results of the selected checkpoint.

Failed check — **Agent Name:**
Passed check — **OK**
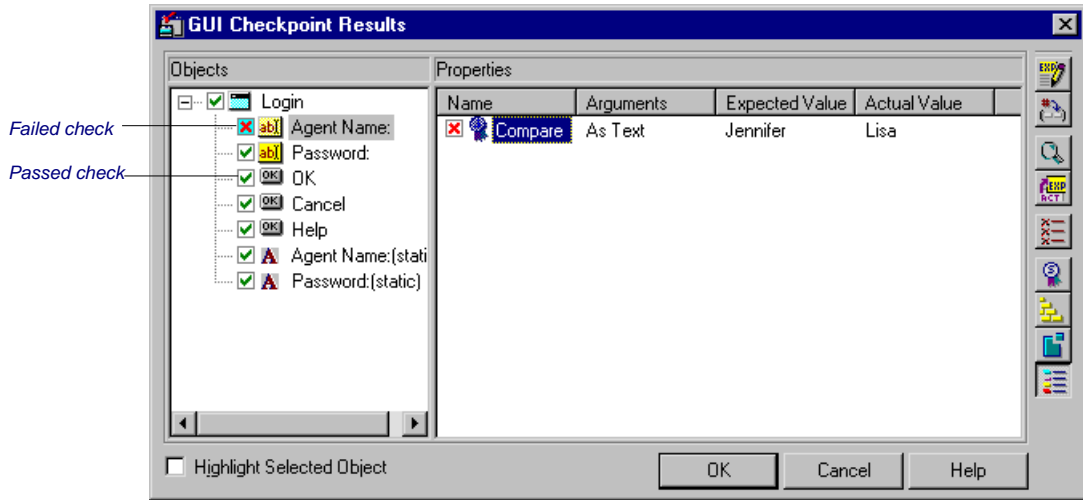
The dialog box lists every object checked and the types of checks performed. Each check is marked as either passed or failed and the expected and the actual results are shown.

You can update the expected value of a checkpoint. For additional information on see **Updating the Expected Results of a Checkpoint** on page 736. For a description of other options in this dialog box, see **Options in the GUI Checkpoint Results Dialog Box** on page 712.

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

Options in the GUI Checkpoint Results Dialog Box

The GUI Checkpoint Results dialog box includes the following options:

Button	Description
	Edit Expected Value enables you to edit the expected value of the selected property. For more information, see **Editing the Expected Value of a Property** on page 263.
	Specify Arguments enables you to specify the arguments for a check on the selected property. For more information, see **Specifying Arguments for Property Checks** on page 252.
	Compare Expected and Actual Values opens the Compare Values box, which displays the expected and actual values for the selected property check. For a check on table contents, opens the Data Comparison Viewer, which displays the expected and actual values for the check.
	Update Expected Value updates the expected value to the actual value. Note that this overwrites the saved expected value.
	Show Failures Only displays only failed checks.
	Show Standard Properties Only displays only standard properties.
	Show Nonstandard Properties Only displays only nonstandard properties, such as Visual Basic, PowerBuilder, and ActiveX control properties.

Button	Description
	Show User Properties Only displays only user-defined property checks. To create user-defined property checks, refer to the *WinRunner Customization Guide.*
	Show All Properties displays all properties, including standard, nonstandard, and user-defined properties.

Viewing the Results of a GUI Checkpoint on Table Contents

You can view the results of a GUI checkpoint on table contents. The results of a GUI checkpoint are displayed 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 on checking the contents of a table, see Chapter 12, **Checking Table Contents**.

To display the results of a GUI checkpoint on table contents:

 1 Choose **Tools > Test Results**, or click the **Test Results** button in the main WinRunner window to open the **Test Results** window.

Books Online

Find

Find Again

Help

Top of Chapter

Back
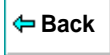
2 Double-click an "end GUI checkpoint" entry in the test log. Alternatively, highlight the entry and choose **Options > Display** or click the **Display** button. The **GUI Checkpoint Results** dialog box opens and the results of the selected GUI checkpoint are displayed.

Objects	Properties			
⊟ ☒ 🖳 Flights Table	Name	Arguments	Expected Value	Actual Value
└ ☒ 🐾 grdFlightTable	☑ 🔍 X		18	18
	☑ 🔍 Y		42	42
	☑ 📊 Height		1695.000000	1695.000000
	☒ 📊 TableCo...		<complex val...	<complex val...
	☑ 📊 Width		5820.000000	5820.000000

☑ Highlight Selected Object   OK   Cancel   Help
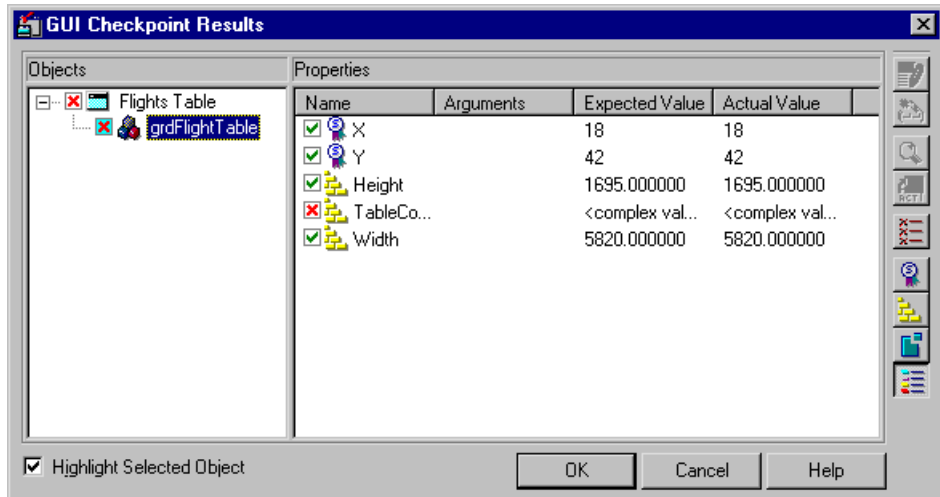
3 Highlight the **TableContent** check and click the **Display** button, or double-click the **TableContent** check. Note that the table contents property check may not be called **TableContent**, and may have a different name instead, depending on which toolkit is used.

The **Data Comparison Viewer** opens, displaying 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.

Cell was not included in the comparison.

List of errors and mismatches

Books Online

Find

Find Again

Help

Top of Chapter

Back

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).

4 By default, scrolling between the Expected Data and Actual Data tables in the Data Comparison Viewer is synchronized. When you click a cell, the corresponding cell in the other table flashes red.

To scroll through the tables separately, clear the **Utilities > Synchronize Scrolling** command or click the **Synchronize Scrolling** button to deselect it. Use the scroll bar as needed to view hidden parts of the table.

Books Online

Find

Find Again

Help

Top of Chapter

Back

5 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 view the previous mismatch:** Click the **Previous Mismatch** button.

- **To view the next mismatch:** Click the **Next Mismatch** button.

- **To see all mismatches:** Choose **Utilities > List All Mismatches** or click the **List All Mismatches** button.

- **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.

Note: You can edit the data in the Edit Check dialog box, which you open from the GUI Checkpoint Results dialog box. To do so, highlight the **TableContent** (or corresponding) property check, and click the **Edit Expected Value** button. For information on working with the Edit Check dialog box, see **Understanding the Edit Check Dialog Box** on page 312.

6 Choose **File > Exit** to close the Data Comparison Viewer.

Viewing the Expected Results of a GUI Checkpoint on Table Contents

You can view the expected results of a GUI checkpoint on table contents either before or after you run your test. The expected results of a GUI checkpoint are displayed in the GUI Checkpoint Results dialog box, which you open from the Test Results window. When you view the expected results of a GUI checkpoint on table contents from the Test Results window, you must choose the expected ("exp") mode in the Results box.

Note that you can also view the expected results of a GUI checkpoint on a table from the Edit Check dialog box. For additional information, see Chapter 12, **Checking Table Contents**.

To display the expected results of a GUI checkpoint on table contents:

1 Choose **Tools > Test Results**, or click the **Test Results** button in the main WinRunner window to open the **WinRunner Test Results** window.

2 If "exp" does not already appear as the results folder in the Results box, then select it.

Results box

Note that since you are viewing the *expected* results of a test, the total number of GUI checkpoints performed by WinRunner is zero.

Books Online

Find

Find Again

Help

Top of Chapter

Back
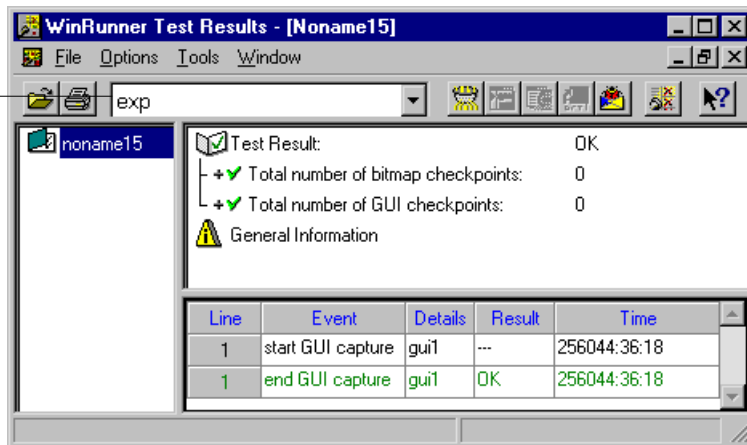
3 Double-click an "end GUI capture" entry in the test log. Alternatively, highlight the entry and choose **Options > Display** or click the **Display** button. The **GUI Checkpoint Results** dialog box opens and the expected results of the selected GUI checkpoint are displayed.
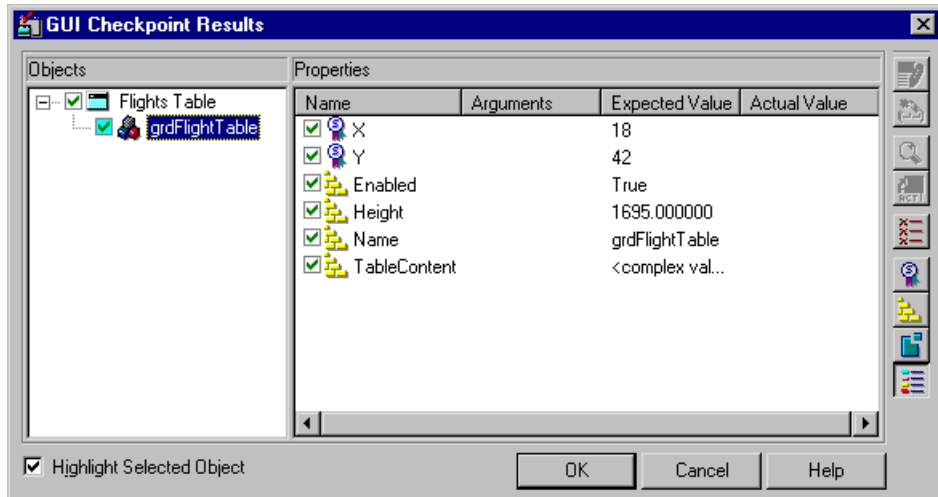
Note: Since you are viewing the *expected* results of the GUI checkpoint, the *actual* values are not displayed.

4 Highlight the **TableContent** check and click the **Display** button, or double-click the **TableContent** check. Note that the table contents property check may not be called **TableContent**, and may have a different name instead, depending on which toolkit is used.

The **Expected Data Viewer** opens, displaying the expected results.

Note: You can edit the data in the Edit Check dialog box, which you open from the GUI Checkpoint Results dialog box. To do so, highlight the **TableContent** (or corresponding) property check, and click the **Edit Expected Value** button. For information on working with the Edit Check dialog box, see **Understanding the Edit Check Dialog Box** on page 312.

5 Choose **File > Exit** to close the Expected Data Viewer.

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 Results** window, look for entries that list bitmap checkpoints in the **Event** column in the test log.

Books Online

Find

Find Again

Help

Top of Chapter

Back

2 To view the results of a specific bitmap checkpoint, double-click its entry in the log. Alternatively, highlight the "bitmap checkpoint" entry and choose **Options > Display** or click the **Display** button. For a mismatch during a test run in Verification or Debug mode, the expected, actual, and difference bitmaps are displayed. For a mismatch during a test run in Update mode, only the expected bitmaps are displayed.

Expected   *Actual*   *Difference*
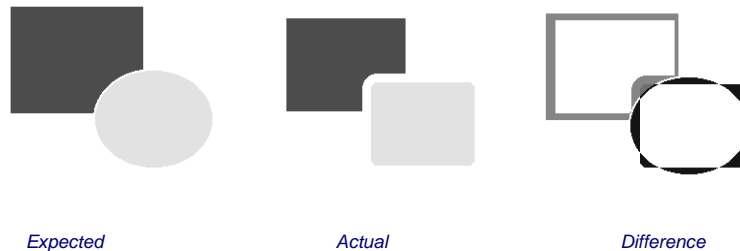
Note: You can control which types of bitmaps are displayed (expected, actual, difference) when you view the results of a bitmap checkpoint. To set the controls, choose Options > Bitmap Controls in the Test Results window.

3 To remove a bitmap from the screen, double-click the system menu button in the bitmap window.

Viewing the Results of a Database Checkpoint

A database checkpoint helps you to identify changes in the contents and structure of databases in your application. The results of a database checkpoint are displayed in the Database Checkpoint Results dialog box that you open from the Test Results window. It displays the database included in the database 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 property checks on the database fail, the entire database checkpoint is marked as failed in the test log.

For more information, see Chapter 13, **Checking Databases**.

To display the results of a database checkpoint:

1 Choose **Tools > Test Results** or click the **Test Results** button in the main WinRunner window to open the **Test Results** window.

2 In the test log, look for entries that list "end database checkpoint" in the **Event** column. Failed database checkpoints appear in red; passed database checkpoints appear in green.

3 Double-click an "end database checkpoint" entry in the test log. Alternatively, highlight the entry and choose **Options > Display** or click the **Display** button. The **Database Checkpoint Results** dialog box opens.

The Database Checkpoint Results dialog box lists the results of the selected checkpoint.

Passed check
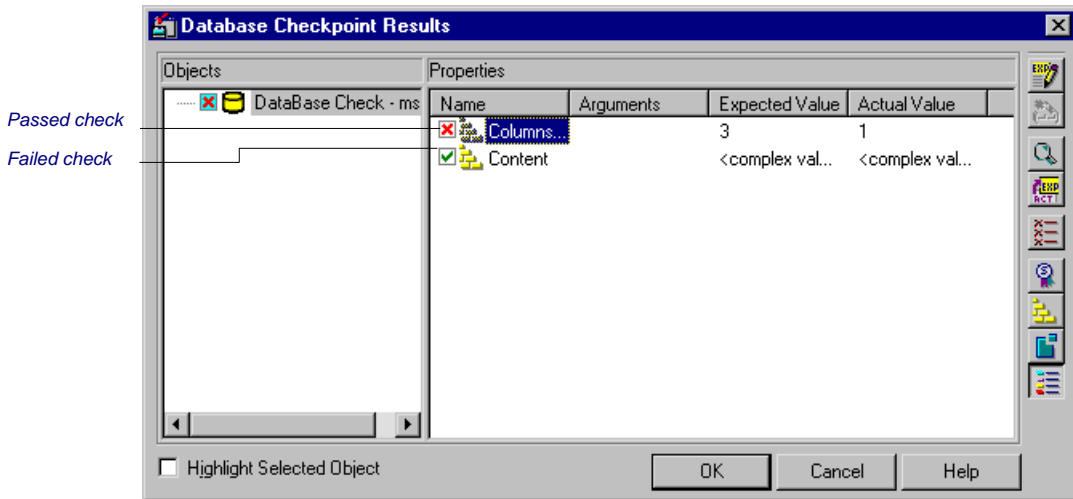
Failed check

The dialog box displays the checked database and the types of checks performed on it. Each check is marked as either passed or failed and the expected and the actual results are shown.

You can update the expected value of a checkpoint. For additional information on see **Updating the Expected Results of a Checkpoint** on page 736. For a description of other options in this dialog box, see **Options in the Database Checkpoint Results Dialog Box** on page 730.

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

Note: You can edit the data in the Edit Check dialog box, which you open from the Database Checkpoint Results dialog box. To do so, highlight the **Content** check, and click the **Edit Expected Value** button. For information on working with the Edit Check dialog box, see **Understanding the Edit Check Dialog Box** on page 359.

Options in the Database Checkpoint Results Dialog Box

The Database Checkpoint Results dialog box includes the following options:

Button	Description
	Edit Expected Value enables you to edit the expected value of the selected property. For more information, see **Creating a Custom Check on a Database** on page 337.
	Compare Expected and Actual Values opens the Compare Values box, which displays the expected and actual values for the selected property check. For a **Content** check, opens the Data Comparison Viewer, which displays the expected and actual values for the check.
	Update Expected Value updates the expected value to the actual value. Note that this overwrites the saved expected value.
	Show Failures Only displays only failed checks.
	Show Standard Properties Only displays only standard properties.
	Show Nonstandard Properties Only displays only nonstandard properties, such as Visual Basic, PowerBuilder, and ActiveX control properties.
	Show All Properties displays all properties, including standard, nonstandard, and user-defined properties.

Viewing the Expected Results of a Content Check in a Database Checkpoint

You can view the expected results of a content check in a database checkpoint either before or after you run your test. The expected results of a database checkpoint are displayed in the Database Checkpoint Results dialog box, which you open from the Test Results window. When you view the expected results of a content check in a database checkpoint from the Test Results window, you must choose the expected ("exp") mode in the Results box.

Note that you can also view the expected results of a database checkpoint on a table from the Edit Check dialog box. For additional information, see Chapter 13, **Checking Databases**.

Books Online

Find

Find Again

? Help

Top of Chapter

Back

To display the expected results of a content check in a database checkpoint:

 1 Choose **Tools > Test Results**, or click the **Test Results** button in the main WinRunner window to open the **WinRunner Test Results** window.

2 If "exp" does not already appear as the results folder in the Results box, then select it.

Results box

Books Online

M Find

Find Again

? Help

Top of Chapter

⇐ Back
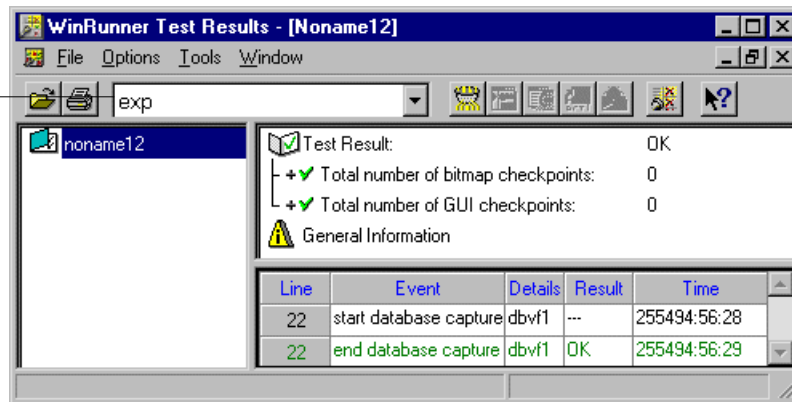
Note that since you are viewing the *expected* results of a test, the total number of database checkpoints performed by WinRunner is zero.

3 Double-click an "end database capture" entry in the test log. Alternatively, highlight the entry and choose **Options > Display** or click the **Display** button. The **Database Checkpoint Results** dialog box opens and the expected results of the selected database checkpoint are displayed.
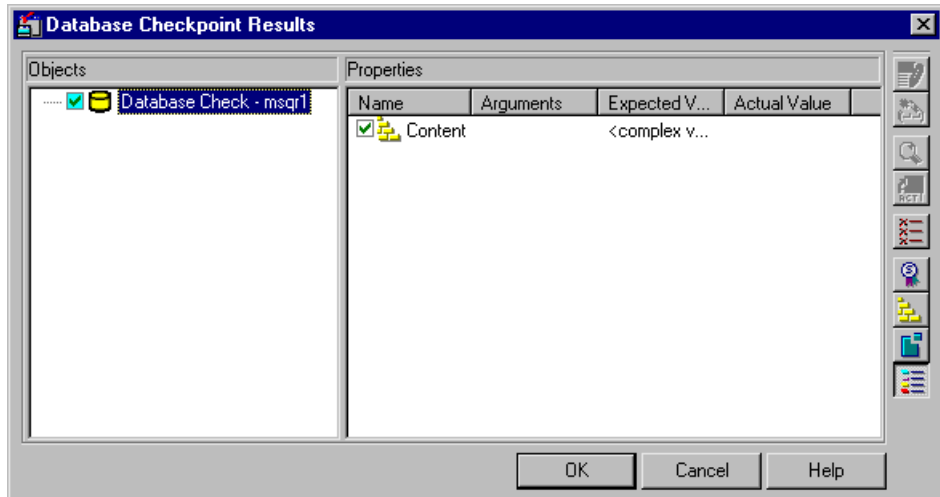
Note that since you are viewing the *expected* results of the database checkpoint, the *actual* values are not displayed.

4 Highlight the **Content** check and click the **Display** button, or double-click the **Content** check.

The **Expected Data Viewer** opens, displaying the expected results.

	Flight	From	Departure	To	Arrival	Ariline	Price
1	9117	DEN	06:33 PM	LAX	07:31 PM	DA	$111.60
2	7220	DEN	05:21 PM	LAX	06:19 PM	DA	$135.20
3	6593	DEN	04:09 PM	LAX	05:07 PM	DA	$124.80
4	6399	DEN	07:45 AM	LAX	08:43 AM	DA	$122.40
5	6240	DEN	03:12 PM	LAX	06:12 PM	AA	$175.47
6	6236	DEN	12:48 PM	LAX	03:48 PM	AA	$102.90
7	6232	DEN	10:24 AM	LAX	01:24 PM	AA	$104.00
8	5696	DEN	11:21 AM	LAX	12:19 PM	DA	$122.40
9	5306	DEN	12:33 PM	LAX	01:31 PM	DA	$120.00
10	2516	DEN	01:45 PM	LAX	02:43 PM	DA	$134.00

Note: You can edit the data in the Edit Check dialog box, which you open from the Database Checkpoint Results dialog box. To do so, highlight the **Content** check, and click the **Edit Expected Value** button. For information on working with the Edit Check dialog box, see **Understanding the Edit Check Dialog Box** on page 359.

5 Choose **File > Exit** to close the Expected Data Viewer.

Books Online

Find

Find Again

Help

Top of Chapter

Back
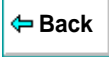
Updating the Expected Results of a Checkpoint

If a bitmap, GUI, or database checkpoint fails, you can update the data in the
expected results folder (*exp*). The next time you run the test, the new expected
results will be compared to the current results in the application.

Updating the Expected Results of a Bitmap Checkpoint

You can update the expected results of a bitmap checkpoint to the actual results
after a test run.

1 In the **Test Results** window, highlight a mismatched "bitmap checkpoint" entry in
the test log.

2 Choose **Options > Update** or click the **Update** button.

3 A dialog box warns that overwriting expected results cannot be undone. Click
Yes to update the results.

Updating the Expected Results of a GUI Checkpoint

You can update the expected results of a GUI checkpoint to the actual results after a test run. You can update the results for the entire GUI checkpoint, or update the results for a specific check within the GUI checkpoint.

To update the expected results for an entire GUI checkpoint:

1 In the **Test Results** window, highlight a mismatched "GUI checkpoint" entry in the test log.

2 Choose **Options > Update** 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 for a specific check within a GUI checkpoint:

1 Double-click the GUI checkpoint entry in the log, choose **Options > Display**, or click the **Display** button.

The **GUI Checkpoint Results** dialog box opens.

2 In the **Properties** pane, highlight a failed check.

3 Click the **Update Expected Value** button.

4 A dialog box warns that if you replace the expected results with the actual results, WinRunner will overwrite the saved expected values. Click **Yes** to update the results.

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

Books Online

M Find

Find Again

? Help

Top of Chapter

⬅ Back

Updating the Expected Results of a Database Checkpoint

You can update the expected results of a database checkpoint to the actual results after a test run. You can update the results for the entire database checkpoint, or update the results for a specific check within a database checkpoint.

To update the expected results for an entire database checkpoint:

 1 In the **Test Results** window, highlight a mismatched "database checkpoint" entry in the test log.

 2 Choose **Options > Update** 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 for a specific check within a database checkpoint:

 1 Double-click the database checkpoint entry in the log, choose **Options > Display**, or click the **Display** button.

   The **Database Checkpoint Results** dialog box opens.

 2 In the **Properties** pane, highlight a failed check.

 3 Click the **Update Expected Value** button.

 4 A dialog box warns that if you replace the expected results with the actual results, WinRunner will overwrite the saved expected values. Click **Yes** to update the results.

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

**Books
Online**

Find

**Find
Again**

Help

**Top of
Chapter**

Back

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 Choose **Tools > Test Results** or click the **Test Results** button to open the **Test Results** window.

2 Double-click a "file compare" event in the test log. Alternatively, highlight the event and choose **Options > Display** or click **Display**. The WDiff utility window opens.

Line contains a mismatch

Line does not contain a mismatch

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 **View > Next Diff** or press the Tab key. The window scrolls to the next highlighted line. To see the previous difference, choose **View > 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 **Options > View > Hide Matching Areas**. 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 a TestDirector project. To report a defect, click the **Report Bug** button or choose **Tools > Report Bug**.

- If the TestDirector Web Defect Manager was previously used on the machine, it opens. The Web Defect Manager is Mercury Interactive's system for reporting and tracking software defects and errors over the World Wide Web. For additional information, see Chapter 42, **Reporting Defects**, or refer to the *Web Defect Manager User's Guide*.

- If the Remote Defect Reporter was installed from the TestDirector CD-ROM, 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 42, **Reporting Defects** for more information.

Books Online

Find

Find Again

Help

Top of Chapter

Back

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:

- **Creating a Batch Test**
- **Running a Batch Test**
- **Storing Batch Test Results**
- **Viewing Batch Test Results**

Books Online

Find

Find Again

Help

Top of Chapter

Back

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.

```
                        ┌──────────────┐
                        │  Batch Test  │
                        └──────┬───────┘
          ┌────────────┬───────┼───────────┬────────────┐
    ┌──────────┐ ┌──────────┐ ┌──────────┐         ┌──────────┐
    │  Test    │ │  Test    │ │  Test    │ - - - - │  Test    │
    │   1      │ │   2      │ │   3      │         │   n      │
    └──────────┘ └──────────┘ └──────────┘         └──────────┘
```
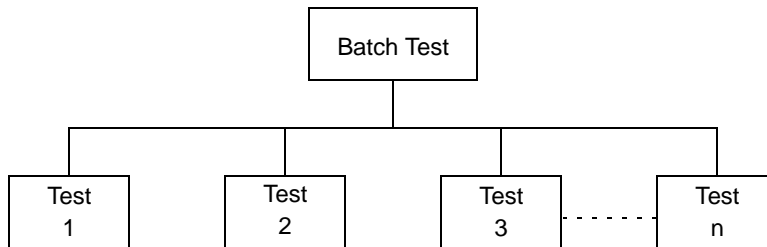
A batch test looks like a regular test that includes call statements. A test becomes a "batch test" when you select the Run in Batch Mode option in the Run tab of the General Options dialog box before you execute the test.

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.

Books Online

M Find

Find Again

? Help

Top of Chapter

⇐ Back

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 30, **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 General Options dialog box before you execute the test.

A batch test may include programming elements such as loops and decision-making statements. Loops enable a batch test to run called tests a specified number of times. Decision-making statements such as *if/else* and *switch* condition test execution on the results of a test called previously by the same batch script. See Chapter 19, **Enhancing Your Test Scripts with Programming**, for more information.

For example, the following batch test executes three tests in succession, then loops back and calls the tests again. The loop specifies that the batch test should call the tests ten times.

```
for (i=0; i<10; i++)
    {
    call "c:\\pbtests\\open" ();
    call "c:\\pbtests\\setup" ();
    call "c:\\pbtests\\save" ();
    }
```

To enable a batch test:

1 Choose **Settings > General Options**.

The **General Options dialog box** opens.

2 Click the **Run** tab.

3 Select the **Run in batch mode** check box.

General Options				
Text Recognition	Environment	Folders	Current Test	
Record	Run		Miscellaneous	

Delay for window synchronization: `100` milliseconds

Timeout for checkpoints and CS statements: `10` milliseconds

Threshold for difference between bitmaps: `0` pixels

☑ Run in batch mode ──────────────── *Run in batch mode check box*

☑ Beep when checking a window

☑ Break when verification fails

☐ Fail test when Context Sensitive errors occur

Advanced...

| OK | Cancel | Apply | Help |

Books Online

Find

Find Again

? Help

◀ ▶

Top of Chapter

⬅ Back

4 Click **OK** to close the General Options dialog box.

For more information on setting the batch option in the General Options dialog box, see Chapter 36, **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 > Run from Top. See Chapter 27, **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 folder for each test. See **Storing Batch Test Results** below for more information. When the batch test run is completed, you can view the test results in the Test Results window.

Note that if your tests contain TSL **texit** statements, WinRunner interprets these statements differently for a batch test run than for a regular test run. During a regular test run, **texit** terminates test execution. During a batch test run, **texit** halts execution of the current test only and control is returned to the batch test.

Storing Batch Test Results

When you run a regular, interactive test, results are stored in a subfolder under the test. The same is true when a test is called by a batch test. WinRunner saves the results for each called test separately in a subfolder under the test. A subfolder 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 an *exp* subfolder under the test folder. Suppose you also create a batch test that calls the three tests. Before running the batch test in Verify mode, you instruct WinRunner to save the results in a folder called *res1*. When the batch test is run, it compares the current test results to the expected results saved earlier. Under each test folder, WinRunner creates a subfolder called *res1* in which it saves the verification results for the test. A *res1* folder is also created under the batch test to contain the overall verification results for the entire run.

Books Online

Find

Find Again

Help

Top of Chapter

Back

```
                    Batch Test ———— exp
                         |            res1
         ┌───────────────┼───────────────┐
       Open            Setup            Save
         |               |                |
        exp             exp              exp
        res1            res1             res1
```
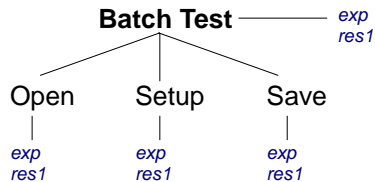
If you run the batch test in Update mode in order to update expected results, WinRunner overwrites the expected results in the *exp* subfolder for each test and for the batch test.

Note that if you run the batch test without selecting the Run in Batch Mode check box in the General Options dialog box, WinRunner saves results only in a subfolder 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 28, **Analyzing Test Results**.

Books Online

Find

Find Again

Help

Top of Chapter

Back
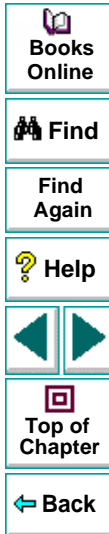
Running Tests
Running Tests from the Command Line

You can run tests directly from the Windows command line.

This chapter describes:

- **Using the Windows Command Line**

- **Command Line Options**

Books Online

Find

Find Again

Help

Top of Chapter

Back

About Running Tests from the Command Line

You can use the Windows Run command to start WinRunner and run a test according to predefined options. You can also save your startup options by creating a custom WinRunner shortcut. Then, to start WinRunner with the startup options, you simply double-click the icon.

Using the command line, you can:

● start WinRunner

● load the relevant tests

● run the tests

● specify test options

● specify the results directories for the test

Most of the functional options that you can set within WinRunner can also be set from the command line. These include test run options and the directories in which test results are stored. You can also specify a *custom.ini* file that contains these and other environment variables and system parameters.

For example, the following command starts WinRunner, loads a batch test, and runs the test:

C\Program Files\Mercury Interactive\WinRunner\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 options. If you plan to use the same set of options each time you start WinRunner, you can create a custom WinRunner shortcut.

Starting WinRunner from the Command Line

This procedure describes how to start WinRunner from the command line.

To start WinRunner from the Run command:

1 On the Windows **Start** menu, choose **Run**. The **Run** dialog box opens.

2 Type in the path of your WinRunner *wrun.exe* file, and then type in any command line options you want to use.

3 Click **OK** to close the dialog box and start WinRunner.

Adding a Custom WinRunner Shortcut

You can make the options you defined permanent by creating a custom WinRunner shortcut.

To add a custom WinRunner shortcut:

1 Create a shortcut for your wrun.exe file in Windows Explorer or My Computer.

2 Click the right mouse button on the shortcut and choose **Properties**.

3 Click the **Shortcut** tab.

4 In the **Target** box, type in any command line options you want to use after the path of your WinRunner wrun.exe file.

5 Click **OK**.

Command Line Options

Following is a description of each command line option.

-addins *list of add-ins to load*

Instructs WinRunner to load the specified add-ins. In the list, the add-ins are separated by commas. This can be used in conjunction with the **-addins_select_timeout** command line option.

-addins_select_timeout *timeout*

Instructs WinRunner to wait the specified time (in seconds) before closing the **Add-In Manager** dialog box when starting WinRunner. When the timeout is zero, the dialog box is not displayed. This can be used in conjunction with the **-addins** command line option.

-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**)

Books Online

M Find

Find Again

? Help

Top of Chapter

← Back

-auto_load_dir *path*

Determines the folder in which the temporary GUI map file (*temp.gui*) resides. This option is applicable only when auto load is on.

(Default = **M_Home\dat**)

-batch {on | off}

Runs the loaded test in Batch mode.

(Default = **off**)

Note that you may also set this option using the **Run in batch mode** check box in the Run tab of the Options dialog box, described in Chapter 36, **Setting Global Testing Options**.

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 37, **Setting Testing Options from a Test Script**.

-beep {on | off}

Activates or deactivates the WinRunner system beep.

(Default = **on**)

Note that you may also set this option using the corresponding **Beep when checking a window** check box in the Run tab of the Options dialog box, described in Chapter 36, **Setting Global Testing Options**.

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 37, **Setting Testing Options from a Test Script**.

-create_text_report {on | off}

Instructs WinRunner to write test results to a text report, *report.txt*, which is saved in the results folder.

(Default = **off**)

-cs_fail {on | off}

Determines whether WinRunner fails a test when Context Sensitive errors occur. A Context Sensitive error is the failure of a Context Sensitive statement during a test. Context Sensitive errors are often due to WinRunner's failure to identify a GUI object.

For example, a Context Sensitive error will occur if you run a test containing a **set_window** statement with the name of a non-existent window. Context Sensitive errors can also occur when window names are ambiguous. For information about Context Sensitive functions, refer to the *TSL Online Reference*.

(Default = **off**)

Note that you may also set this option using the corresponding **Fail test when Context Sensitive errors occur** check box in the Run tab of the Options dialog box, described in Chapter 36, **Setting Global Testing Options**.

Note that you can use the **setvar** and **getvar** functions to set and retrieve the value of the corresponding *cs_fail* testing option from within a test script, as described in Chapter 37, **Setting Testing Options from a Test Script**.

Books Online

M Find

Find Again

? Help

Top of Chapter

← Back

-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])

Note that you may also set this option using the corresponding **Delay between execution of CS statements** box in the Advanced Run Options dialog box, described in Chapter 36, **Setting Global Testing Options**.

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 37, **Setting Testing Options from a Test Script**.

**Books
Online**

Find

**Find
Again**

Help

**Top of
Chapter**

Back

-delay_msec *non-negative integer*

Directs WinRunner to determine whether a window or object is stable before
capturing it for a bitmap checkpoint or synchronization point. It defines the time
(in milliseconds) that WinRunner waits between consecutive samplings of the
screen. If two consecutive checks produce the same results, WinRunner captures
the window or object. (Formerly -**delay**, which was measured in seconds.)

(Default = **1000** [milliseconds])

(Formerly **-delay**.)

Note: This parameter is accurate to within 20-30 milliseconds.

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 36, **Setting Global Testing Options**.

Note that you can use the **setvar** and **getvar** functions to set and retrieve the
value of the corresponding *delay_msec* testing option from within a test script, as
described in Chapter 37, **Setting Testing Options from a Test Script**.

Books Online

Find

Find Again

Help

Top of Chapter

Back

-dont_quit

Instructs WinRunner not to close after completing the test.

-dont_show_welcome

Instructs WinRunner not to display the Welcome window when starting WinRunner.

-exp *expected results folder name*

Designates a name for the subfolder in which expected results are stored. In a verification run, specifies the set of expected results used as the basis for the verification comparison.

(Default = **exp**)

Note that you may also view this setting using the corresponding **Expected results folder** box in the Current Test tab of the Options dialog box, described in Chapter 36, **Setting Global Testing Options**.

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 37, **Setting Testing Options from a Test Script**.

-f *file name*

Specifies a text file containing command line options. The options can appear on
the same line, or each on a separate line. This option enables you to circumvent
the restriction on the number of characters that can be typed into the Target text
box in the Shortcut tab of the Windows Properties dialog box.

Note: If a command line option appears both in the command line and in the file,
WinRunner uses the settings of the option in the file.

-fontgrp *group name*

Specifies the active font group when WinRunner is started.

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 36,
Setting Global Testing Options.

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 37, **Setting Testing Options from a Test Script**.

-ini *initialization test name*

Defines the *wrun.ini* file that is used when WinRunner is started. This file is read-only, unless the **-update_ini** command line option is also used.

-min_diff *non-negative integer*

Defines the number of pixels that constitute the threshold for an image mismatch.

(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 36, **Setting Global Testing Options**.

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 37, **Setting Testing Options from a Test Script**.

Books Online

Find

Find Again

Help

Top of Chapter

Back

-mismatch_break {on | off}

Activates or deactivates Break when Verification Fails before a verification run. The functionality of Break when Verification Fails is different than when running a test interactively: In an interactive run, the test is paused; For a test started from the command line, the first occurrence of a comparison mismatch terminates the test run.

Break when Verification Fails determines whether WinRunner pauses the test run and displays a message whenever verification fails or whenever any message is generated as a result of a Context Sensitive statement during a test that is run in Verify mode.

For example, if a **set_window** statement is missing from a test script, WinRunner cannot find the specified window. If this option is on, WinRunner pauses the test and opens the Run wizard to enable the user to locate the window. If this option is off, WinRunner reports an error in the Test Results window and proceeds to run the next statement in the test script.

(Default = **off**)

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 36, **Setting Global Testing Options**.

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 37, **Setting Testing Options from a Test Script**.

Books Online

Find

Find Again

Help

Top of Chapter

Back

-rec_item_name {0 | 1}

Determines whether WinRunner records non-unique ListBox and ComboBox items by name or by index.

(Default = **0**)

Note that you may also set this option using the corresponding **Record non-unique list items by name** check box in the Record tab of the General Options dialog box, described in Chapter 36, **Setting Global Testing Options**.

Note that you can use the **setvar** and **getvar** functions to set and retrieve the value of the corresponding *rec_item_name* testing option from within a test script, as described in Chapter 37, **Setting Testing Options from a Test Script**.

-run

Instructs WinRunner to run the loaded test. To load a test into the WinRunner window, use the **-t** command line option.

Books Online

Find

Find Again

Help

Top of Chapter

Back

-run_minimized

Instructs WinRunner to open minimized. Note that specifying this option does not itself run tests: use the **-t** command line option to load a test and the **-run** command line option to run the loaded test.

-search_path *path*

Defines the directories to be searched for tests to be opened and/or called. The search path is given as a string.

(Default = **startup folder** and **installation folder\lib**)

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 36, **Setting Global Testing Options**.

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 37, **Setting Testing Options from a Test Script**.

-single_prop_check_fail {0 | 1}

Fails a test run when **_check_info** statements fail. It also writes an event to the Test Results window for these statements. (You can create **_check_info** statements using the **Create > GUI Checkpoint > For Single Property** command.)

You can use this option with the **setvar** and **getvar** functions.

(Default = **1**)

For information about the **check_info** functions, refer to the *TSL Online Reference*.

Note that you may also set this option using the corresponding **Fail test when single property check fails** option in the Run tab of the General Options dialog box, described in Chapter 36, **Setting Global Testing Options**.

Note that you can use the **setvar** and **getvar** functions to set and retrieve the value of the corresponding *single_prop_check_fail* testing option from within a test script, as described in Chapter 37, **Setting Testing Options from a Test Script**.

Books Online

Find

Find Again

Help

Top of Chapter

Back

-speed {normal | fast}

Sets the speed for the execution of the loaded test.

(Default = **fast**)

Note that you may also set this option using the corresponding **Run Speed for Analog Mode** option in the Advanced Run Options dialog box, described in Chapter 36, **Setting Global Testing Options**.

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 37, **Setting Testing Options from a Test Script**.

-t *test name*

Specifies the name of the test to be loaded in the WinRunner window. This can be the name of a test stored in a folder specified in the search path or the full pathname of any test stored in your system.

Books Online

Find

Find Again

Help

Top of Chapter

Back

-td_connection {on | off}

Activates or deactivates WinRunner's connection to TestDirector.

(Default = **off**)

Note that you can use the corresponding *td_connection* testing option to activate or deactivate WinRunner's connection to TestDirector, as described in Chapter 37, **Setting Testing Options from a Test Script**.

Note that you can connect to TestDirector from the **Connection to TestDirector** dialog box, which you open by choosing Tools > TestDirector Connection. For more information about connecting to TestDirector, see Chapter 40, **Managing the Testing Process**.

-td_cycle_name *cycle name*

Specifies the name of the current test cycle. This option is applicable only when WinRunner is connected to TestDirector.

Note that you can use the corresponding *td_cycle_name* testing option to specify the name of the current test cycle, as described in Chapter 37, **Setting Testing Options from a Test Script**.

-td_database_name *database path*

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.

Note that you can use the corresponding *td_database_name* testing option to specify the active TestDirector database, as described in Chapter 37, **Setting Testing Options from a Test Script**.

Note that when WinRunner is connected to TestDirector, you can specify the active TestDirector project database from the **Connection to TestDirector** dialog box, which you open by choosing **Tools > TestDirector Connection**. For more information, see Chapter 40, **Managing the Testing Process**.

-td_logname_dir *event log file path*

Defines the full pathname for an event log file. Note that this file is not a TestDirector file.

-td_password *password*

Specifies the password for connecting to a database in a TestDirector server.

Note that you can specify the password for connecting to TestDirector from the **Connection to TestDirector** dialog box, which you open by choosing **Tools > TestDirector Connection**. For more information about connecting to TestDirector, see Chapter 40, **Managing the Testing Process**.

-td_server_name *server name*

Specifies the name of the TestDirector server to which WinRunner connects.

Note that you can use the corresponding *td_server_name* testing option to specify the name of the TestDirector server to which WinRunner connects, as described in Chapter 37, **Setting Testing Options from a Test Script**.

Note that you can specify the name of the TestDirector server to which WinRunner connects from the **Connection to TestDirector** dialog box, which you open by choosing **Tools > TestDirector Connection**. For more information about connecting to TestDirector, see Chapter 40, **Managing the Testing Process**.

-td_user_name *user name*

Specifies the name of the user who is currently executing a test cycle.

(Formerly **-user**.)

Note that you can use the corresponding *td_user_name* testing option to specify the user, as described in Chapter 37, **Setting Testing Options from a Test Script**.

Note that you can specify the user name when you connect to TestDirector from the **Connection to TestDirector** dialog box, which you open by choosing **Tools > TestDirector Connection**. For more information about connecting to TestDirector, see Chapter 40, **Managing the Testing Process**.

-timeout_msec *non-negative integer*

Defines the global *timeout_msec* variable (in milliseconds) used by WinRunner. (Formerly *timeout*, which was measured in seconds.)

(Default = **1000** [milliseconds])

(Formerly **-timeout**.)

Note: This option is accurate to within 20-30 milliseconds.

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 36, **Setting Global Testing Options**.

Note that you can use the **setvar** and **getvar** functions to set and retrieve the value of the corresponding *timeout_msec* testing option from within a test script, as described in Chapter 37, **Setting Testing Options from a Test Script**.

-tslinit_exp *expected results folder*

Directs WinRunner to the expected folder to be used when the *tslinit* script is running.

-update_ini

Saves changes to configuration made during a WinRunner session when the *wrun.ini* file is specified by the **-ini** command line option.

Note: You can only use this command line option when you also use the **-ini** command line option.

-verify *verification results folder name*

Specifies that the test is to be run in Verify mode and designates the name of the subfolder in which the test results are stored.

Debugging Tests

Books
Online

Find

Find
Again

Help

Top of
Chapter

Back

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**

Books Online

Find

Find Again

? Help

Top of Chapter

Back

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 Step 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 32, **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 33, **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* folder. 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 27, **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

Choose the **Step** command or click the corresponding **Step** button to execute only the current line of the active test script—the line marked by the execution arrow.

When the current line calls another test or a user-defined function, the called test or function is executed in its entirety but the called test script is not displayed in the WinRunner window.

Step Into

Choose the **Step Into** command or click the corresponding **Step Into** button to execute only the current line of the active test script. However, in contrast to Step, if the current line of the executed test calls another test or a user-defined function in compiled mode:

● The test script of the called test or function is displayed in the WinRunner window.

● 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 **Run > Step to Cursor** or press the STEP TO CURSOR softkey. WinRunner runs 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 choosing **Run > Pause**, clicking the **Pause** button, or pressing the PAUSE softkey. A paused test stops running when all previously interpreted TSL statements have been executed. Unlike the Stop command, Pause does not initialize test variables and arrays.

To resume execution of a paused test, choose the appropriate Run command on 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*.

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:

- **Breakpoint Types**
- **Setting Break at Location Breakpoints**
- **Setting Break in Function Breakpoints**
- **Modifying Breakpoints**
- **Deleting Breakpoints**

Books Online

Find

Find Again

Help

Top of Chapter

Back

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 the test run when it reaches a breakpoint. You can examine the effects of the test run up to the breakpoint, make any necessary changes, and then continue running 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 33, **Monitoring Variables**, for more information.

● mark a point from which to begin stepping through a test script using the Step commands. See Chapter 31, **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.

Books Online

Find

Find Again

Help

Top of Chapter

Back

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 *ui_func* function, in the *ui_test* compiled module: 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 **Debug > Breakpoints** to open the **Breakpoints** dialog box. Alternatively, click the **Break in Function** button or choose **Debug > Break in Function** to open the **New Breakpoint** dialog box and proceed to step 3.

Closes the Breakpoints dialog box.

All currently defined Location breakpoints

Opens the New Breakpoints dialog box.

All currently defined Function breakpoints
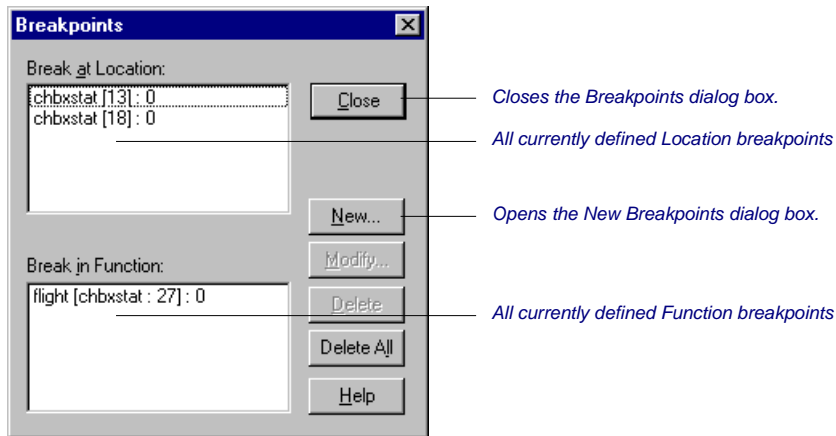
2 Click **New** to open the **New Breakpoint** dialog box.
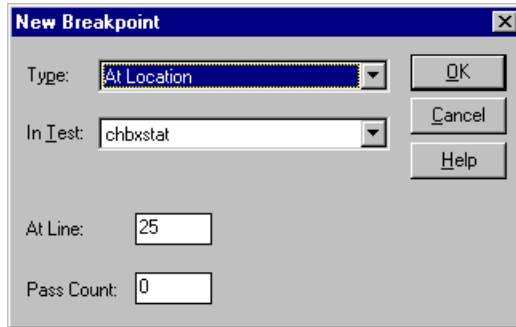
3 Click the breakpoint type and the test name. Modify the line number in the **At Line** box and the pass count in the **Pass Count** box as required.

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 set a Break at Location breakpoint using the Toggle Breakpoint command:

1 Move the insertion point to the line of the test script where you want test execution to stop.

2 Choose **Debug > Toggle Breakpoint** or click the **Toggle Breakpoint** button. The breakpoint symbol (!) appears in the left margin of the WinRunner window.

To remove a Break at Location breakpoint:

1 Click the breakpoint symbol with the right mouse button.

2 Choose **Debug > Toggle Breakpoint**, or click the **Toggle Breakpoint** button.

Setting Break in Function Breakpoints

A Break in Function breakpoint stops test execution at the user-defined function that you specify. You 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 **Debug > Breakpoints** to open the **Breakpoints** dialog box. Alternatively, click the **Break in Function** button and proceed to step 3.

Books Online

Find

Find Again

? Help

◀ ▶

Top of Chapter

⬅ Back

Breakpoints dialog box:

Break at Location:
chbxstat [13] : 0
chbxstat [18] : 0

Closes the Breakpoints dialog box.

All currently defined Location breakpoints

Opens the New Breakpoints dialog box.

Break in Function:
flight [chbxstat : 27] : 0

All currently defined Function breakpoints

Buttons: Close, New..., Modify..., Delete, Delete All, Help
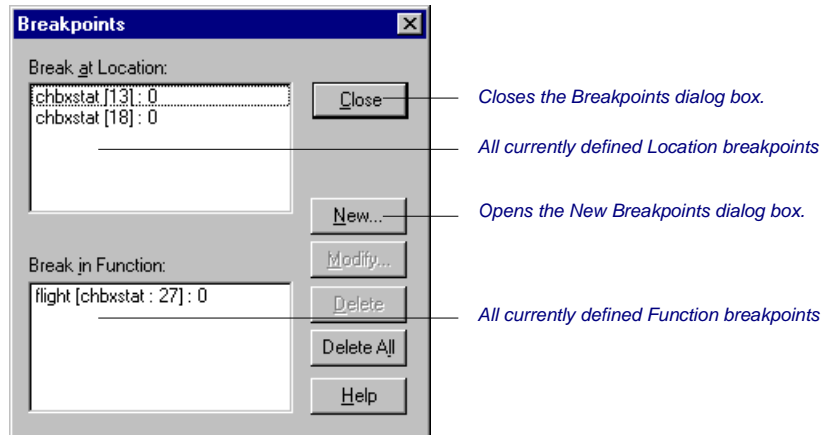
2 Click **New** to open the **New Breakpoint** dialog box.

3 In the **Type** box, click **In Function**. The dialog box changes so that you can type in a function name and a pass count value.
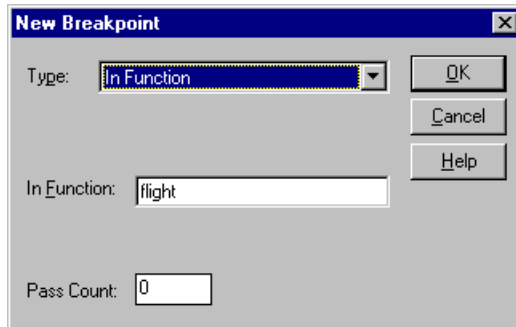
4 Enter the name of a user-defined function in the **In Function** box. The function must be compiled by WinRunner. For more information, see Chapter 22, **Creating User-Defined Functions**, and Chapter 23, **Creating Compiled Modules**.

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.
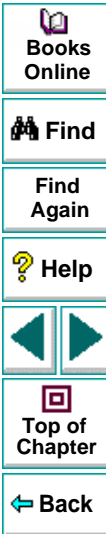
To set a Break in Function breakpoint using the Break in Function command:

 1 Choose **Debug > Break in Function**. The **New Breakpoint** dialog box opens.

New Breakpoint	
Type: In Function ▼	OK
	Cancel
	Help
In Function: flight	
Pass Count: 0	
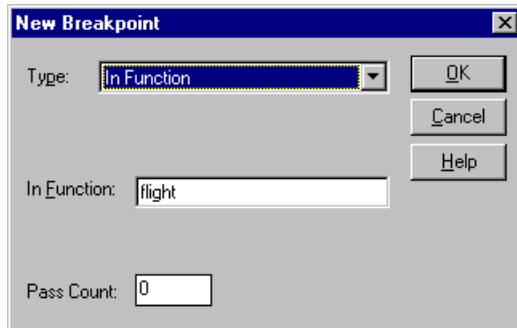
 2 Type the name of a user-defined function in the **In Function** box. The function must be compiled by WinRunner. For more information, see Chapter 22, **Creating User-Defined Functions**, and Chapter 23, **Creating Compiled Modules**.

 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 **Debug > Breakpoints** 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.

Modify Breakpoint		
Type:	At Location	OK
In Test:	chbxstat	Cancel
		Help
At Line:	13	
Pass Count:	0	
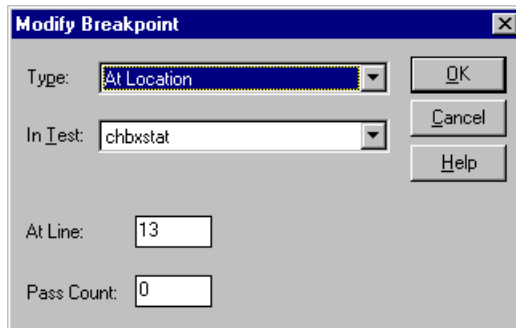
4 To change the type of breakpoint, select a different breakpoint type in the **Type** box.

To select another test, click 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 **Debug > Breakpoints** 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:

- **Adding Variables to the Watch List**
- **Viewing Variables in the Watch List**
- **Modifying Variables in the Watch List**
- **Assigning a Value to a Variable in the Watch List**
- **Deleting Variables from the Watch List**

About Monitoring Variables

The Watch List enables you to monitor the values of variables, expressions, and array elements while you debug a test script. Prior to running a test, you add the elements that you want to monitor to the Watch List. 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 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 you can view the current values.
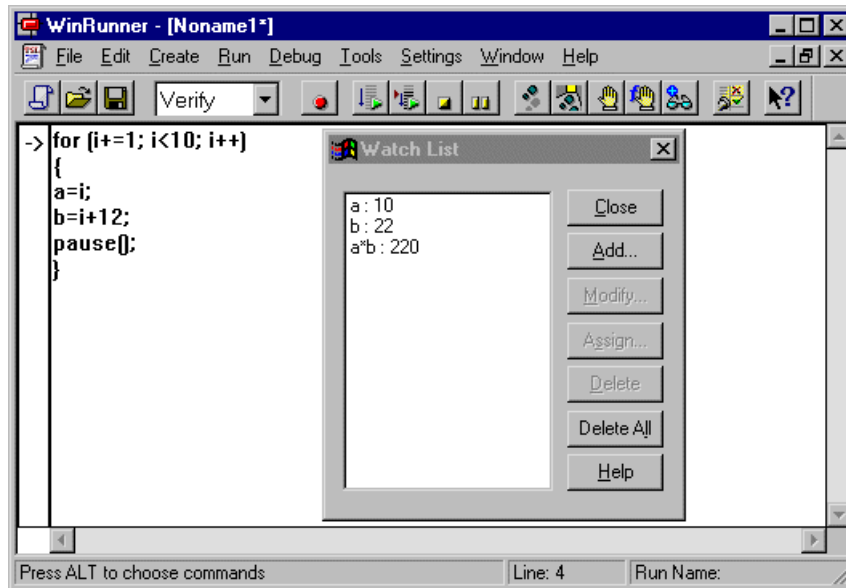
After WinRunner executes the first loop, the test pauses. The Watch List displays the variables and updates their values: When WinRunner completes the test run, the Watch List shows the following results:

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 current value 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.

Books Online

M Find

Find Again

? Help

Top of Chapter

⬅ Back

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 **Debug > Add Watch** or click the **Add Watch** button to open the **Add Watch** dialog box.

Add Watch	
Expression:	OK
a	Cancel
	Help
Value:	
<cannot evaluate>	Evaluate
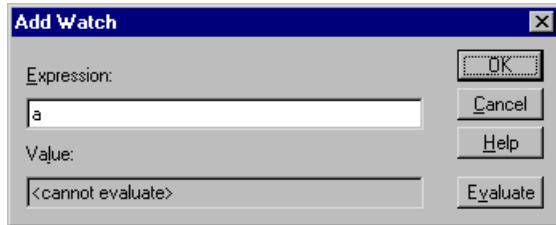
Alternatively, you can open the Add Watch dialog box from the Watch List. Choose **Debug > Watch List** and click **Add**.

 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 you add variables, expressions, and arrays to the Watch List, you can use the Watch List to view their values.

To view the values of variables, expressions, and arrays in the Watch List:

 1 Choose **Debug > Watch List** to open the **Watch List** dialog box.

The variables, expressions and arrays are displayed; current values appear after the colon.

2 To view values of array elements, double-click the array name. The elements and their values appear under the array name. Double-click the array name to hide the elements.

```
Watch List                          ×

a : 10                          Close
b : 22
a*b : 220                        Add...
numbers : <array>
   numbers[0] : 1               Modify...
   numbers[1] : 22
   numbers[2] : 220             Assign...
   numbers[3] : 12
                                Delete

                               Delete All

                                 Help
```
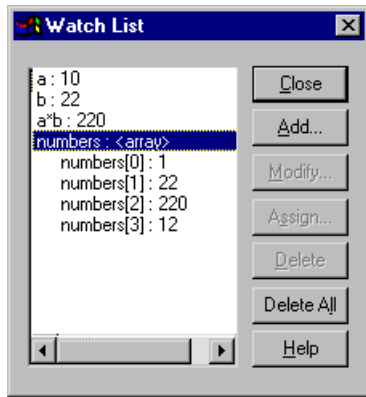
3 Click **Close**.

Modifying Variables in the Watch List

You can modify variables and expressions in the Watch List using the Modify Watch dialog box. For example, you can turn variable *b* into the expression *b + 1*, or you can change the expression *b + 1* into *b * 10*. When you close the Modify Watch dialog box, the Watch List is automatically updated to reflect the new value for the expression.

To modify an expression in the Watch List:

1 Choose **Debug > Watch List** 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.

Modify Watch	
Expression:	OK
b+1	Cancel
	Help
Value:	
13	Evaluate
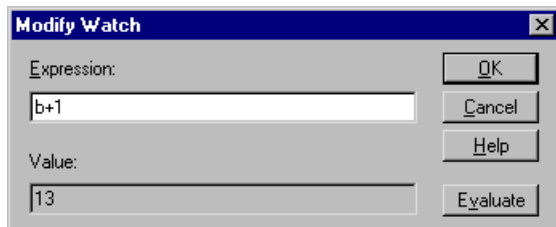
4 Change the expression in the **Expression** box as needed.

5 Click **Evaluate**. The new value of the expression appears in the **Value** box.

6 Click **OK** to close the **Modify Watch** dialog box. The modified expression and its new value appear in the Watch List.

Assigning a Value to a Variable in the Watch List

You can assign new values to variables and array elements in the Watch List. Values can be assigned only to variables and array elements, not to expressions.

To assign a value to a variable or an array element:

1 Choose **Debug > Watch List** 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.
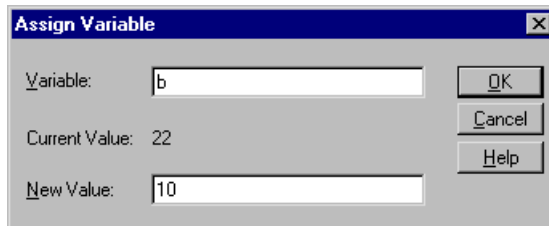
4 Type the new value for the variable or array element in the **New Value** box.

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 **Debug > Watch List** to open the **Watch List** dialog box.

2 Select a variable, an expression, or an array to delete.

Note: You can delete an array only if its elements are hidden. To hide the elements of an array, double-click the array name in the Watch List.

---

3 Click **Delete** 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 Choose **Debug > Watch List** to open the **Watch List** dialog box.

2 Click **Delete All**. All entries are deleted.

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

Books Online

Find

Find Again

Help

Top of Chapter

Back

Configuring WinRunner

Books
Online

Find

Find
Again

Help

Top of
Chapter

Back

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:

● Customizing the User Toolbar

● Using the User Toolbar

● Configuring WinRunner Softkeys

Books Online

M Find

Find Again

? Help

Top of Chapter

← Back

About Customizing WinRunner's User Interface

You can adapt WinRunner's user interface to your testing needs by changing the way you access WinRunner commands.

You may find that when you create and run tests, you frequently use the same WinRunner menu commands and insert the same TSL statements into your test scripts. You can create shortcuts to these commands and TSL statements by customizing the 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.
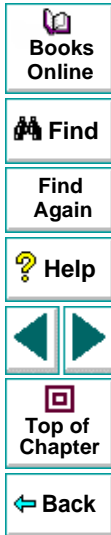
Books Online

M Find

Find Again

? Help

Top of Chapter

⇐ Back

Customizing the User Toolbar

The User toolbar contains buttons for commands used when creating tests. In its default setting, the User toolbar enables easy access to the following WinRunner commands:
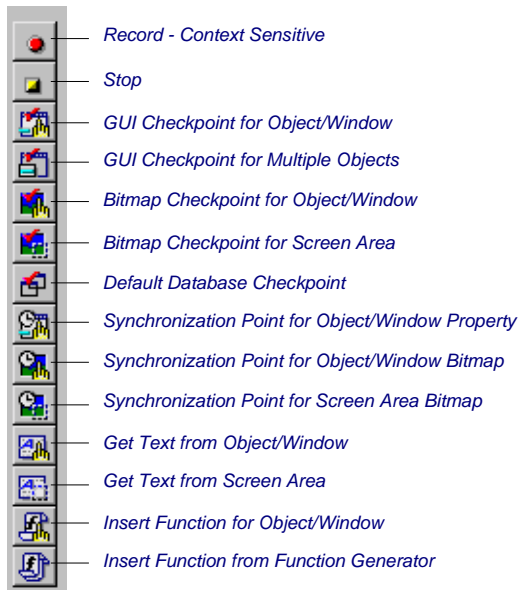
Record - Context Sensitive

Stop

GUI Checkpoint for Object/Window

GUI Checkpoint for Multiple Objects

Bitmap Checkpoint for Object/Window

Bitmap Checkpoint for Screen Area

Default Database Checkpoint

Synchronization Point for Object/Window Property

Synchronization Point for Object/Window Bitmap

Synchronization Point for Screen Area Bitmap

Get Text from Object/Window

Get Text from Screen Area

Insert Function for Object/Window

Insert Function from Function Generator

Books Online

Find

Find Again

Help

Top of Chapter

Back

By default, the User toolbar is hidden. To display the User toolbar, select it on the Window menu. When it is displayed, its default position is docked at the right edge of the WinRunner window.
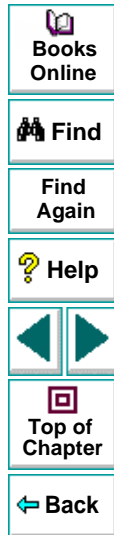
The User toolbar is a customizable toolbar. You can add or remove buttons to facilitate access to the commands you most frequently use when testing an application. You can use the User toolbar to:

Books Online

Find

Find Again

Help

Top of Chapter

Back

● Execute additional WinRunner menu commands. For example, you can add a button to the User toolbar that opens the GUI Map Editor.

● Paste TSL statements into your test scripts. For example, you can add a button to the User toolbar that pastes the TSL statement **report_msg** into your test scripts.

● Execute TSL statements. For example, you can add a button to the User toolbar that executes the TSL statement **load ("my_module");**.

● Parameterize TSL statements before pasting them into your test scripts or executing them. For example, you can add a button to the User toolbar that enables you to add parameters to the TSL statement **list_select_item**, and then either paste it into your test script or execute it.

Edit GUI Map — *Parameterize list_select_item*

Paste report_msg *Execute load ("my_module");*
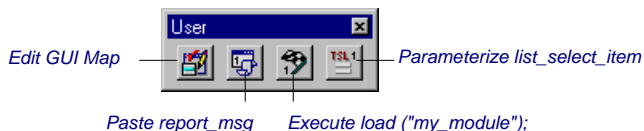
Note: None of the buttons that appear by default in the User toolbar appear in the illustration above.
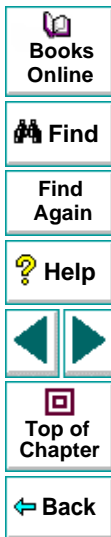
Adding Buttons that Execute Menu Commands

You can add buttons to the User toolbar that execute frequently-used menu commands.

The tables below illustrate the buttons you can add to the User toolbar and the corresponding menu commands. In cases where the name of a button differs from the name of the menu command, the menu command appears in italics below the button name. Buttons that appear on the User toolbar by default are marked with an asterisk (*).

File Menu

Command	Button	Command	Button
New		Save All	
Open		Test Properties	
Save		Print	
Save As			

Edit Menu

Command	Button	Command	Button
Undo		Delete	
Redo		Select All	
Cut		Find	
Copy		Replace	
Paste		Go To	
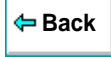
Create Menu

Command	Button	Command	Button
Record - Context Sensitive *		Synchronization Point > For Object/Window Bitmap *	
Stop *		Synchronization Point > For Screen Area Bitmap *	
GUI Checkpoint > For Single Property		Edit GUI Checklist	
GUI Checkpoint > For Object/Window *		Edit Database Checklist	
GUI Checkpoint > For Multiple Objects *		Get Text > From Object/Window *	
Bitmap Checkpoint > For Object/Window *		Get Text > From Screen Area *	
Bitmap Checkpoint > For Screen Area *		Insert Function > For Object/Window *	
Default Database Checkpoint *		Insert Function > From Function Generator *	
Synchronization Point > For Object/Window Property *		RapidTest Script Wizard	

Books Online

Find

Find Again

Help

Top of Chapter

Back
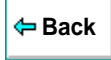
Run Menu

Command	Button	Command	Button
Run from Top		Step Into	
Run from Arrow		Step Out	
Run Minimized (Top) *Run Minimized > From Top*		Step to Cursor	
Run Minimized (Arrow) *Run Minimized > From Arrow*		Pause	
Step		Stop	

Debug Menu

Command	Button	Command	Button
Breakpoints		Watch List	
Toggle Breakpoint		Add Watch	
Break in Function			

Tools Menu

Command	Button	Command	Button
Spy *GUI Spy*		Fonts Expert	
Edit GUI Map *GUI Map Editor*		Exception Handling	
Configure GUI Map *GUI Map Configuration*		TestDirector Connection	
Learn Virtual Objects *Virtual Object Wizard*		Data Table	
ActiveX Properties Viewer		Parameterize Value	
Test Results			

Settings Menu

Command	Button	Command	Button
General Options		Customize User Toolbar	
Editor Options			

Window Menu

Command	Button	Command	Button
Cascade		Arrange Icons	
Tile Horizontally		Close All	
Tile Vertically			

Books Online

Find

Find Again

Help

Top of Chapter

Back

To add a menu command to the User toolbar:

1 Choose **Settings** > **Customize User Toolbar**.

The Customize User Toolbar dialog box opens.
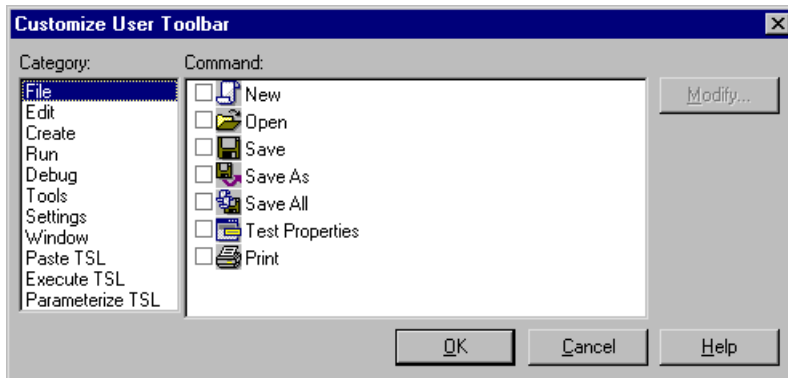
Note that each menu in the menu bar corresponds to a category in the Category pane of the Customize User Toolbar dialog box.

2 In the **Category** pane, select a menu.

3 In the **Command** pane, select the check box next to the menu command.

4 Click **OK** to close the Customize User Toolbar dialog box.

The selected menu command button is added to the User toolbar.

To remove a menu command from the User toolbar:

 1 Choose **Settings** > **Customize User Toolbar** to open the Customize User Toolbar dialog box.

 2 In the **Category** pane, select a menu.

 3 In the **Command** pane, clear the check box next to the menu command.

 4 Click **OK** to close the Customize User Toolbar dialog box.

The selected menu command button is removed from the User toolbar.
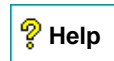
Adding Buttons that Paste TSL Statements

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 **Settings** > **Customize User Toolbar**.

The Customize User Toolbar dialog box opens.

2 In the **Category** pane, select **Paste TSL**.

3 In the **Command** pane, select the check box next to a button, and then select the button.

4 Click **Modify**.

The Paste TSL Button Data dialog box opens.

Books Online

Find

Find Again

Help

Top of Chapter

Back
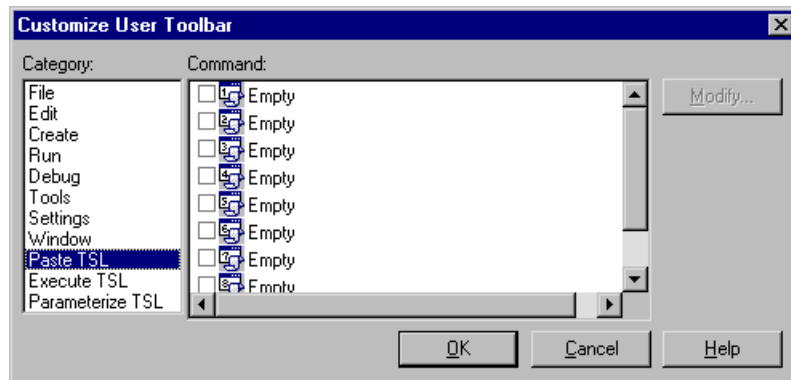
Paste TSL Button Data	⊠
Button Title:	
Text to Paste:	
_OK_ _Cancel_ _Help_	
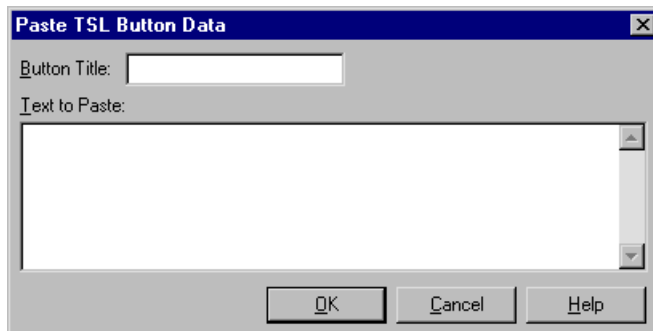
5 In the **Button Title** box, enter a name for the button.

6 In the **Text to Paste** pane, enter the TSL statement(s).

7 Click **OK** to close the Paste TSL Button Data dialog box.

The name of the button is displayed beside the corresponding button in the Command pane.

8 Click **OK** to close the Customize User Toolbar dialog box.

The button is added to the User toolbar.

To modify a button on the User toolbar that pastes TSL statements:

 1 Choose **Settings** > **Customize User Toolbar** to open the Customize User Toolbar dialog box.

 2 In the **Category** pane, select **Paste TSL**.

 3 In the **Command** pane, select the button whose content you want to modify.

 4 Click **Modify**.

   The Paste TSL Button Data dialog box opens.

 5 Enter the desired changes in the **Button Title** box and/or the **Text to Paste** pane.

 6 Click **OK** to close the Paste TSL Button Data dialog box.

 7 Click **OK** to close the Customize User Toolbar dialog box.

   The button on the User toolbar is modified.

To remove a button from the User toolbar that pastes TSL statements:

 1 Choose **Settings** > **Customize User Toolbar** to open the Customize User Toolbar dialog box.

 2 In the **Category** pane, select **Paste TSL**.

 3 In the **Command** pane, clear the check box next to the button.

 4 Click **OK** to close the Customize User Toolbar dialog box.

The button is removed from the User toolbar.
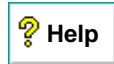
Adding Buttons that Execute TSL Statements

You can add buttons to the User toolbar that execute frequently-used TSL statements.

To add a button to the User toolbar that executes a TSL statement:

1 Choose **Settings** > **Customize User Toolbar**.

The Customize User Toolbar dialog box opens.

2 In the **Category** pane, select **Execute TSL**.

3 In the **Command** pane, select the check box next to a button, and then select the button.

4 Click **Modify**.

The Execute TSL Button Data dialog box opens.

Execute TSL Button Data

TSL Statement:

$$\underline{O}K \qquad \underline{C}ancel \qquad \underline{H}elp$$
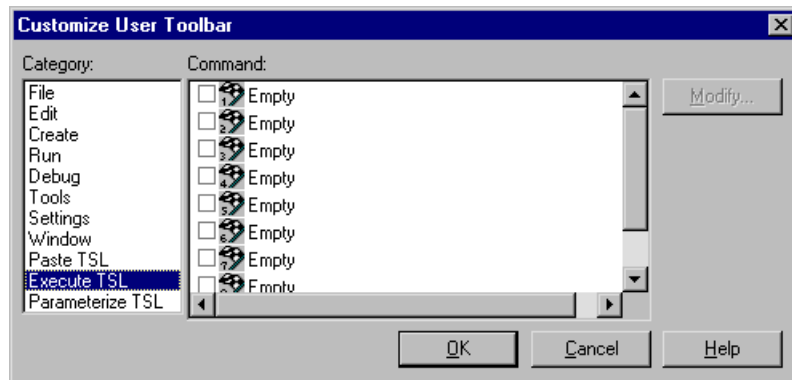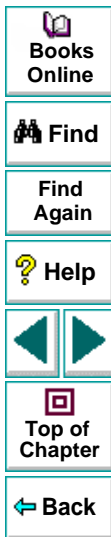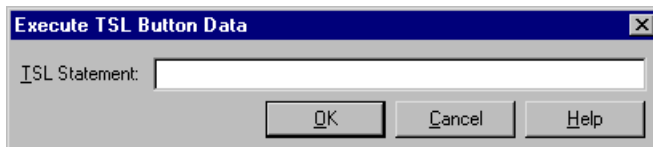
5 In the **TSL Statement** box, enter the TSL statement.

6 Click **OK** to close the Execute TSL Button Data dialog box.

The TSL statement is displayed beside the corresponding button in the Command pane.

7 Click **OK** to close the Customize User Toolbar dialog box.

The button is added to the User toolbar.

To modify a button on the User toolbar that executes a TSL statement:

1 Choose **Settings** > **Customize User Toolbar** to open the Customize User Toolbar dialog box.

2 In the **Category** pane, select **Execute TSL**.

3 In the **Command** pane, select the button whose content you want to modify.

4 Click **Modify**.

The Execute TSL Button Data dialog box opens.

5 Enter the desired changes in the **TSL Statement** box.

6 Click **OK** to close the Execute TSL Button Data dialog box.

7 Click **OK** to close the Customize User Toolbar dialog box.

The button on the User toolbar is modified.

To remove a button from the User toolbar that executes a TSL statement:

1 Choose **Settings** > **Customize User Toolbar** to open the Customize User Toolbar dialog box.

2 In the **Category** pane, select **Execute TSL**.

3 In the **Command** pane, clear the check box next to the button.

4 Click **OK** to close the Customize User Toolbar dialog box.

The button is removed from the User toolbar.

Adding Buttons that Parameterize TSL Statements

You can add buttons to the User toolbar that enable you to easily parameterize frequently-used TSL statements, and then paste them into your test script or execute them.

To add a button to the User toolbar that enables you to parameterize a TSL statement:

1 Choose **Settings** > **Customize User Toolbar**. The Customize User Toolbar dialog box opens.

2 In the **Category** pane, select **Parameterize TSL**.

3 In the **Command** pane, select the check box next to a button, and then select the button.

Books Online

M Find

Find Again

? Help

Top of Chapter

Back

4 Click **Modify**.

The Parameterize TSL Button Data dialog box opens.

Parameterize TSL Button Data	×
TSL Statement:	
	OK Cancel Help

Books Online

Find

Find Again

Help
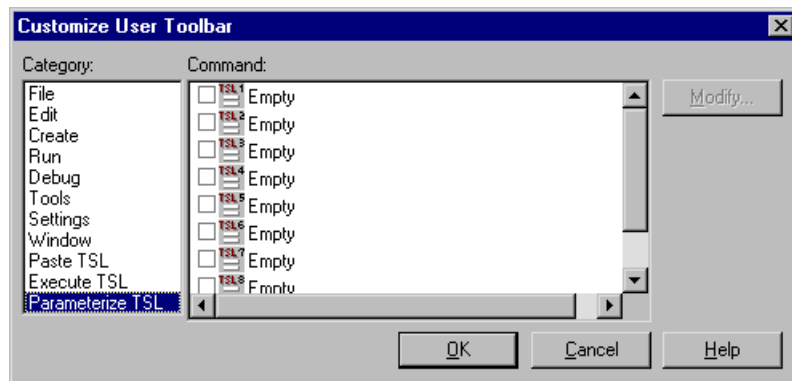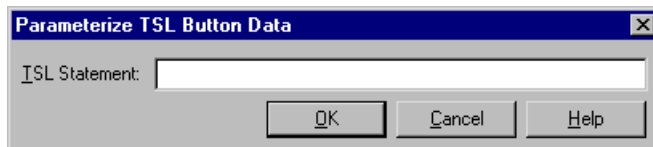
5 In the **TSL Statement** box, enter the name of TSL function. You do not need to enter any parameters. For example, enter **list_select_item**.

6 Click **OK** to close the Parameterize TSL Button Data dialog box.

The TSL statement is displayed beside the corresponding button in the Command pane.

7 Click **OK** to close the Customize User Toolbar dialog box.

The button is added to the User toolbar.

Top of Chapter

Back

To modify a button on the User toolbar that enables you to parameterize a TSL statement:

1 Choose **Settings** > **Customize User Toolbar** to open the Customize User Toolbar dialog box.

2 In the **Category** pane, select **Parameterize TSL**.

3 In the **Command** pane, select the button whose content you want to modify.

4 Click **Modify**.

 The Parameterize TSL Button Data dialog box opens.

5 Enter the desired changes in the **TSL Statement** box.

6 Click **OK** to close the Parameterize TSL Button Data dialog box.

7 Click **OK** to close the Customize User Toolbar dialog box.

 The button on the User toolbar is modified.

Books Online

Find

Find Again

Help

Top of Chapter

Back

To remove a button from the User toolbar that enables you to parameterize a TSL statement:

 1 Choose **Settings** > **Customize User Toolbar** to open the Customize User Toolbar dialog box.

 2 In the **Category** pane, select **Parameterize TSL**.

 3 In the **Command** pane, clear the check box next to the button.

 4 Click **OK** to close the Customize User Toolbar dialog box.

The button is removed from the User toolbar.

Books Online

Find

Find Again

Help

Top of Chapter

Back

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 TSL-based commands that appear on the User toolbar by choosing them on the Create menu.

When the User toolbar is a "floating" toolbar, it remains open when you minimize WinRunner while recording a test. For additional information, see Chapter 8, **Creating Tests**.

Books Online

M Find

Find Again

? Help

Top of Chapter

← Back
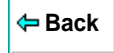
Parameterizing a TSL Statement

When you click a button on the User toolbar that represents a TSL statement to be parameterized, the Set Function Parameters dialog box opens.
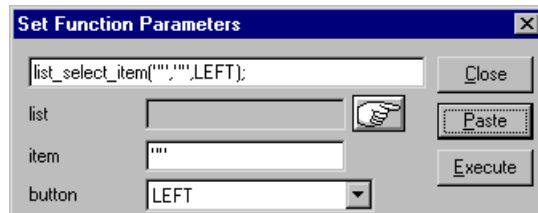
The Set Function Parameters dialog box varies in its appearance according to the parameters required by a particular TSL function. For example, the **list_select_item** function has three parameters: *list*, *item*, and *button*. For each parameter, you define a value as described below:

- To define a value for the *list* parameter, you click the pointing hand. WinRunner is minimized, 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, you type it in the corresponding box.

- To define a value for the *button* parameter, you select it from the list.

Books
Online

M Find

**Find
Again**

? Help

**Top of
Chapter**

← Back
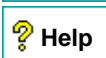
Accessing TSL Statements on the Menu Bar

All TSL statements that you add to the User toolbar can also be accessed via the Create menu.

To choose a TSL statement from a menu:

- To paste a TSL statement, you click **Create** > **Paste TSL** > [TSL Statement].

- To execute a TSL statement, you click **Create** > **Execute TSL** > [TSL Statement].

- To parameterize a TSL statement, you click **Create** > **Parameterize TSL** > [TSL Statement].

Books Online

Find

Find Again

Help

Top of Chapter

Back

Configuring WinRunner Softkeys

Several WinRunner commands can be carried out using softkeys. WinRunner can carry out softkey commands even when the WinRunner window is not the active window on your screen, or when it is minimized.

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

Default Settings for WinRunner Softkeys

The following table lists the default softkey configurations and their functions.

Command	Default Softkey Combination	Function
RECORD	F2	Starts test recording. While recording, this softkey toggles between Context Sensitive and Analog modes.
CHECK GUI FOR SINGLE PROPERTY	Alt Right + F12	Checks a single property of a GUI object.
CHECK GUI FOR OBJECT/WINDOW	Ctrl Right + F12	Creates a GUI checkpoint for an object or a window.

Books Online

Find

Find Again

Help

Top of Chapter

Back

Command	Default Softkey Combination	Function
CHECK GUI FOR MULTIPLE OBJECTS	F12	Opens the Create GUI Checkpoint dialog box.
CHECK BITMAP OF OBJECT/WINDOW	Ctrl Left + F12	Captures an object or a window bitmap.
CHECK BITMAP OF SCREEN AREA	Alt Left + F12	Captures an area bitmap.
CHECK DATABASE (DEFAULT)	Ctrl Right + F9	Creates a check on the entire contents of a database.
CHECK DATABASE (CUSTOM)	Alt Right + F9	Checks the number of columns, rows and specified information of a database.
SYNCHRONIZE OBJECT/WINDOW PROPERTY	Ctrl Right + F10	Instructs WinRunner to wait for a property of an object or a window to have an expected value.
SYNCHRONIZE BITMAP OF OBJECT/WINDOW	Ctrl Left + F11	Instructs WinRunner to wait for a specific object or window bitmap to appear.
SYNCHRONIZE BITMAP OF SCREEN AREA	Alt Left + F11	Instructs WinRunner to wait for a specific area bitmap to appear.

Books Online

Find

Find Again

Help

Top of Chapter

Back

Command	Default Softkey Combination	Function
GET TEXT FROM OBJECT/WINDOW	F11	Captures text in an object or a window.
GET TEXT FROM WINDOW AREA	Alt Right + F11	Captures text in a specified area and adds an **obj_get_text** statement to the test script.
GET TEXT FROM SCREEN AREA	Ctrl Right + F11	Captures text in a specified area and adds a **get_text** statement to the test script.
INSERT FUNCTION FOR OBJECT/WINDOW	F8	Inserts a TSL function for a GUI object.
INSERT FUNCTION FROM FUNCTION GENERATOR	F7	Opens the Function Generator dialog box.
RUN FROM TOP	Ctrl Left + F5	Runs the test from the beginning.
RUN FROM ARROW	Ctrl Left + F7	Runs the test from the line in the script indicated by the arrow.
STEP	F6	Runs only the current line of the test script.

Books Online

Find

Find Again

Help

Top of Chapter

Back

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 the test run after all previously interpreted TSL statements have been executed. Execution can be resumed from this point using the Run from Arrow command or the RUN FROM ARROW softkey.
STOP	Ctrl Left + F3	Stops test recording or the test run.
MOVE LOCATOR	Alt Left + F6	Records a move_locator_abs statement with the current position (in pixels) of the screen pointer.

Books Online

M Find

Find Again

? Help

Top of Chapter

Back

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 **Start** > **Programs** > **WinRunner** > **Softkey Configuration**. The Softkey Configuration dialog box opens.

The Commands pane lists all the WinRunner softkeys.
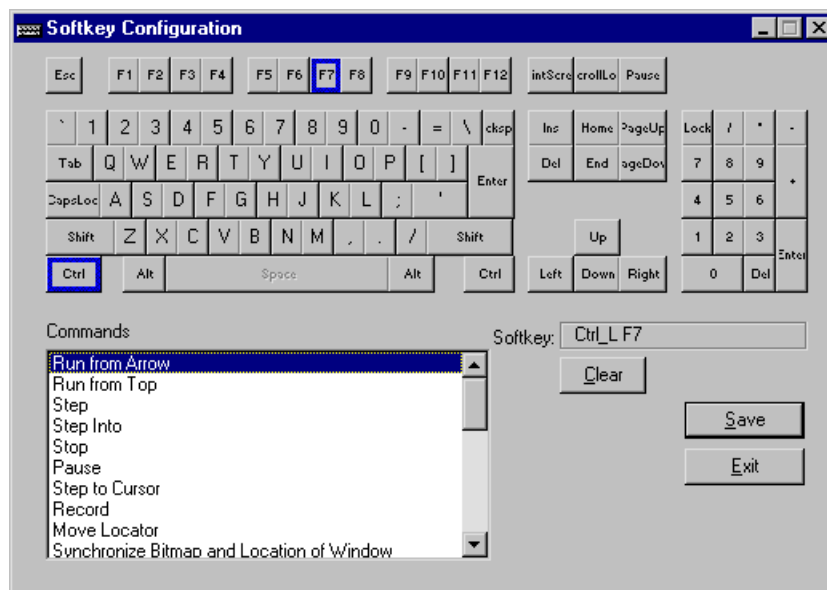
2 Click the command you want to change. The current softkey definition appears in the **Softkey** box; its keys are highlighted on the keyboard.

3 Click the new key or combination that you want to define. The new definition appears in the **Softkey** box.

An error message appears if you choose a definition that is already in use or an illegal key combination. Click a different key or combination.

4 Click **Save** to save the changes and close the dialog box. The new softkey configuration takes effect when you start WinRunner.

WinRunner includes a powerful and customizable script editor. This enables you to set the size of margins in test windows, change the way the elements of a test script appear, and create a list of typing errors that will be automatically corrected by WinRunner.

This chapter describes:

- **Setting Display Options**
- **Personalizing Editing Commands**

Books Online

Find

Find Again

Help

Top of Chapter

Back

About Customizing the Test Script Editor

WinRunner's script editor lets you set display options, and personalize script editing commands.

Setting Display Options

Display options let you configure WinRunner's test windows and how your test scripts will be displayed. For example, you can set the size of test window margins, and activate or deactivate word wrapping.

Display options also let you change the color and appearance of different script elements. These include comments, strings, WinRunner reserved words, operators and numbers. For each script element, you can assign colors, text attributes (bold, italic, underline), font, and font size. For example, you could display all strings in the color red.

Finally, there are display options that let you control how the hard copy of your scripts will appear when printed.

Personalizing Script Editing Commands

WinRunner includes a list of default keyboard commands that let you move the cursor, delete characters, cut, copy, and paste information to and from the clipboard. You can replace these commands with commands you prefer. For example, you could change the Set Bookmark [#] command from the default CTRL + K + [#] to CTRL + B + [#].

Setting Display Options

WinRunner's display options let you control how test scripts appear in test windows, how different elements of test scripts are displayed, and how test scripts will appear when they are printed.

Customizing Test Scripts and Windows

You can customize the appearance of WinRunner's test windows and how your scripts are displayed. For example, you can set the size of the test window margins, highlight script elements, and show or hide text symbols.

Books Online

Find

Find Again

? Help

Top of Chapter

Back

To customize the appearance of your script:

 1 Choose **Settings > Editor Options**. The Editor Options dialog box opens.

Editor Options

Options | Highlighting | Key assignments |

Options

Print Options
- ☐ Wrap long lines
- ☐ Line numbers
- ☑ Title in header
- ☑ Date in header
- ☑ Page numbers

General options
- ☑ Auto indent
- ☐ Smart tab
- ☑ Smart fill
- ☑ Use tab character
- ☐ Line numbers in gutter
- ☑ Statement completion

- ☐ Show all chars
- ☑ Block cursor for Overwrite
- ☑ Word select
- ☑ Syntax highlight
- ☐ Cursor beyond EOL

- ☐ Visible right margin
- ☑ Visible gutter

Right margin 0

Gutter width 37

Block indent step size 4   Tab stop 4

OK   Cancel   Help

 2 Click the **Options** tab.

3 Under the **General options** choose from the following options:

Options	Description
Auto indent	Causes lines following an indented line to automatically begin at the same point as the previous line. You can click the Home key on your keyboard to move the cursor back to the left margin.
Smart tab	A single press of the tab key will insert the appropriate number of tabs and spaces in order to align the cursor with the text in the line above.
Smart fill	Insert the appropriate number of tabs and spaces in order to apply the Auto indent option. When this option is not selected, only spaces are used to apply the Auto indent. (Both **Auto indent** and **Use tab character** must be selected to apply this option).
Use tab character	Inserts a tab character when the tab key on the keyboard is used. When this option is not enabled, the appropriate number of space characters will be inserted instead.
Line numbers in gutter	Displays a line number next to each line in the script. The line number is displayed in the test script window's gutter.

Options	Description
Statement completion	Opens a list box displaying all available matches to the function prefix whenever the user presses the Ctrl and Space keys simultaneously or the Underscore key. Select an item from the list to replace the typed string. To close the list box, press the Esc key. Displays a tooltip with the function parameters once the complete function name appears in the editor.
Show all chars	Displays all text symbols, such as tabs and paragraph symbols.
Block cursor for Overwrite	Displays a block cursor instead of the standard cursor when you select overwrite mode.
Word select	Selects the nearest word when you double-click on the test window.
Syntax highlight	Highlights script elements such as comments, strings, or reserved words. For information on reserved words, see **Reserved Words** on page 848.
Cursor beyond EOL	Enables WinRunner to display the cursor after the end of the current line.
Visible right margin	Displays a line that indicates the test window's right margin.
Right margin	Sets the position, in characters, of the test window's right margin (0=left window edge).

Books Online

Find

Find Again

Help

Top of Chapter

Back

Options	Description
Visible gutter	Displays a blank area (gutter) in the test window's left margin.
Gutter width	Sets the width, in pixels, of the gutter.
Block indent step size	Sets the number characters that the selected block of TSL statements will be moved (indented) when the INDENT SELECTED BLOCK softkey is used. For information about editor softkeys, see **Personalizing Editing Commands** on page 851.
Tab stop	Sets the distance, in characters, between each tab stop.

Highlighting Script Elements

WinRunner scripts contain many different elements, such as comments, strings, WinRunner reserved words, operators and numbers. Each element of a WinRunner script is displayed in a different color and style. You can create your own personalized color scheme and style for each script element. For example, all comments in your scripts could be displayed as italicized, blue letters on a yellow background.
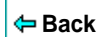
To edit script elements:

 1 Choose **Settings > Editor Options**. The Editor Options dialog box opens to the
   Highlighting tab.

Books Online

Find

Find Again

Help

Top of Chapter
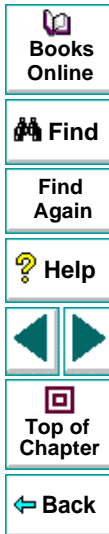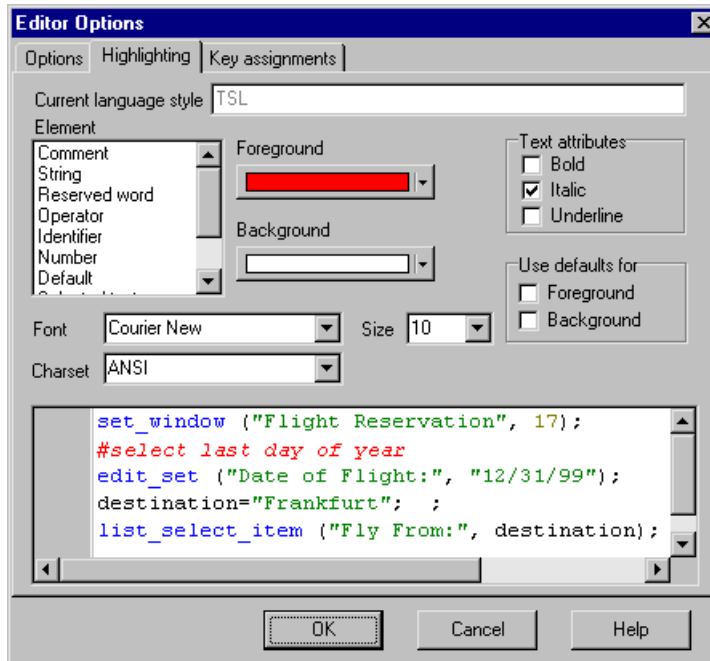
Back

2 Select a script element from the **Elements** list.

3 Choose from the following options:

Options	Description
Foreground	Sets the color applied to the text of the script element.
Background	Sets the color that appears behind the script element.
Text Attributes	Sets the text attributes applied to the script element. You can select bold, italic, or underline or a combination of these attributes.
Use defaults for	Applies the font and colors of the "default" style to the selected style.
Font	Sets the typeface of the script element.
Size	Set the size, in points, of the script element.
Charset	Sets the character subset of the selected font.

Books Online

Find

Find Again

Help

Top of Chapter

Back

An example of each change you apply will be displayed in the pane at the bottom of the dialog box.

4 Click **OK** to apply the changes.

Reserved Words

WinRunner contains "reserved words," which include the names of all TSL functions and language keywords, such as auto, break, char, close, continue, int, function. For a complete list of all reserved words in WinRunner, refer to the *TSL Online Reference*. You can add your own reserved words in the *[ct_KEYWORD_USER]* section of the *reserved_words.ini* file, which is located in the *dat* folder in the WinRunner installation directory. Use a text editor, such as Notepad, to open the file. Note that after editing the list, you must restart WinRunner so that it will read from the updated list.

Customizing Print Options

You can set how the hard copy of your script will appear when it is sent to the printer. For example, your printed script can include line numbers, the name of the file, and the date it was printed.

To customize your print options:

1 Choose **Settings > Editor Options**. The Editor Options dialog box opens.

2 Click the **Options** tab.

Editor Options

Options | Highlighting | Key assignments

Options

Print Options
- [ ] Wrap long lines
- [ ] Line numbers
- [x] Title in header
- [x] Date in header
- [x] Page numbers

General options
- [x] Auto indent
- [ ] Smart tab
- [x] Smart fill
- [x] Use tab character
- [ ] Line numbers in gutter
- [x] Statement completion

- [ ] Show all chars
- [x] Block cursor for Overwrite
- [x] Word select
- [x] Syntax highlight
- [ ] Cursor beyond EOL

- [ ] Visible right margin   [x] Visible gutter
- Right margin [0]   Gutter width [37]

Block indent step size [4]   Tab stop [4]

OK   Cancel   Help

3 Choose from the following Print options:

Option	Description
Wrap long lines	Automatically wraps a line of text to the next line if it is wider than the current printer page settings.
Line numbers	Prints a line number next to each line in the script.
File name in header	Inserts the file name into the header of the printed script.
Date in header	Inserts today's date into the header of the printed script.
Page numbers	Numbers each page of the script.

4 Click **OK** to apply the changes.

Books Online

Find

Find Again

? Help

Top of Chapter

Back

Personalizing Editing Commands

You can personalize the default keyboard commands you use for editing test scripts. WinRunner includes keyboard commands that let you move the cursor, delete characters, cut, copy, and paste information to and from the clipboard. You can replace these commands with your own preferred commands. For example, you could change the Paste command from the default CTRL + V to CTRL + P.

To personalize editing commands:

1 Choose **Settings > Editor Options**. The Editor Options dialog box opens.

 2 Click the **Key Assignments** tab.

Books Online

Find

Find Again

Help

Top of Chapter

Back
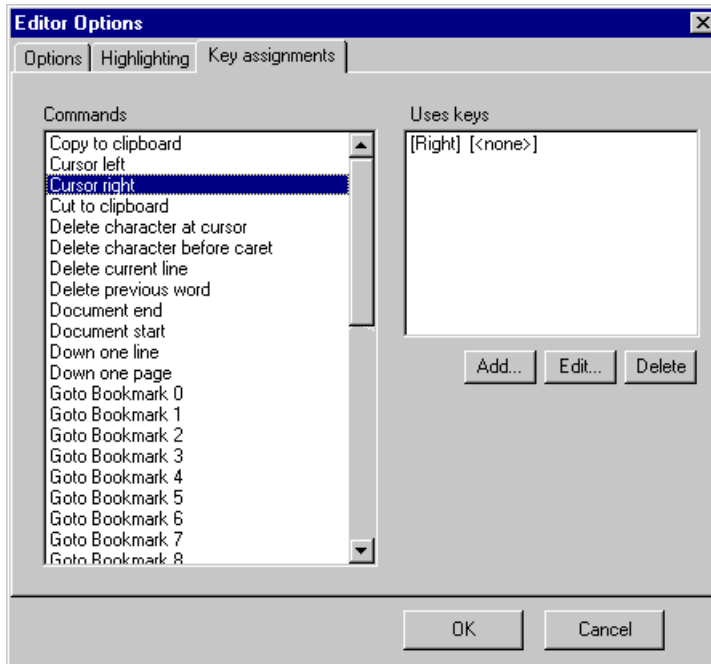
 3 Select a command from the **Commands** list.
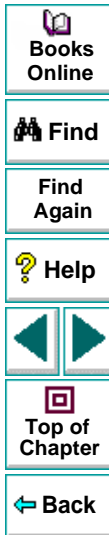
4 Click **Add** to create an additional key assignment or click **Edit** to modify the existing assignment. The Add/Edit key pair for dialog box opens. Press the keys you want to use. For example CTRL + 4.

> **Add new key pair for: Undo** ⊠
>
> **Step 1**
>
> Press the first key combination until the correct key sequence appears then press the NEXT button
> [Ctrl+4]
> [Clear]
>
> [< Back] [Next >] [Cancel]
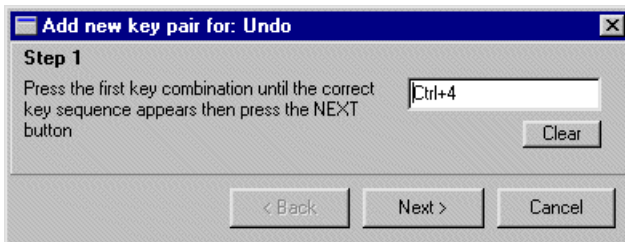
5 Click **Next**. To add an additional key sequence, press the keys you want to use. For example U.

> **Add new key pair for: Undo** ⊠
>
> **Step 2**
>
> If the key sequence requires a 2nd key press then enter the key now. Press the FINISH button to accept changes
> [U]
> [Clear]
>
> [< Back] [Finish] [Cancel]
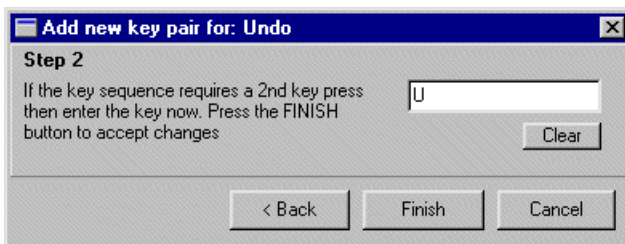
6 Click **Finish** to add the key sequence(s) to the **Use keys** list.

If you want to delete a key sequence from the list, highlight the keys in the **Uses Key** list and click **Delete**.

7 Click **OK** to apply the changes.

Configuring WinRunner
Setting Global Testing Options

You can control how WinRunner records and runs tests by setting global testing options from the General Options dialog box.

This chapter describes:

- **Setting Global Testing Options from the General Options Dialog Box**
- **Global Testing Options**

About Setting Global Testing Options

WinRunner testing options affect how you record test scripts and. 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 General 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 37, **Setting Testing Options from a Test Script**.

Setting Global Testing Options from the General Options Dialog Box

Before you record or run tests, you can use the General 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.

Books Online

M Find

Find Again

? Help

Top of Chapter

Back

To set global testing options:

1 Choose **Settings > General Options**.

The **General Options** dialog box opens. It is divided by subject into seven tabbed pages.
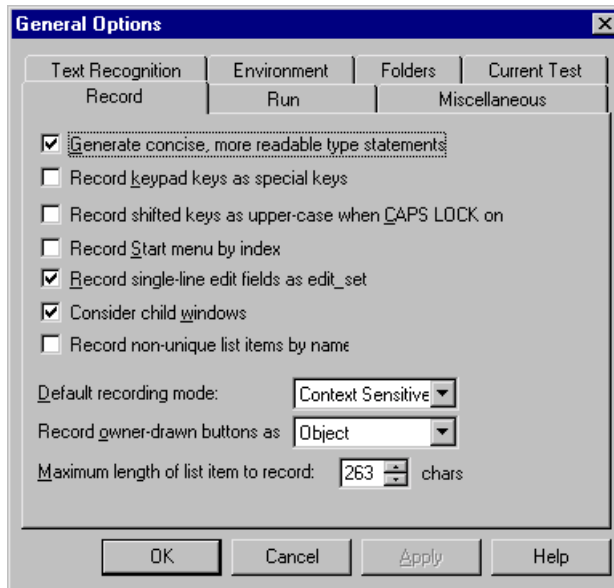
2 To choose a page, click a tab.

3 Set an option, as described in **Global Testing Options** on page 858.

4 To apply your change and keep the General Options dialog box open, click **Apply**.

5 When you are done, click **OK** to apply your change and close the dialog box.
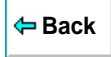
Global Testing Options

The General 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 General 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 37, **Setting Testing Options from a Test Script**.

Record Tab

The Record tab options affect how WinRunner records tests.

General 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

☐ Record non-unique list items by name

Default recording mode: Context Sensitive ▾

Record owner-drawn buttons as Object ▾

Maximum length of list item to record: 263 ⬍ chars

| OK | Cancel | Apply | Help |

Books Online

Find

Find Again

Help

Top of Chapter

Back

Generate Concise, More Readable 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 37, **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.

Books Online

Find

Find Again

Help

Top of Chapter

Back

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*.

Books Online

Find

Find Again

Help

Top of Chapter

Back

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 does not identify 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 37, **Setting Testing Options from a Test Script**.

Record Non-Unique List Items by Name

This option determines how WinRunner records non-unique ListBox and ComboBox items.

☑ When this option is selected, WinRunner records non-unique items by name.

☐ When this option is cleared, WinRunner records non-unique items by index.

(Default = **cleared**)

Note that you can use the **setvar** and **getvar** functions to set and retrieve the value of the corresponding *rec_item_name* testing option from within a test script, as described in Chapter 37, **Setting Testing Options from a Test Script**.

Note that you can also set this option using the corresponding *-rec_item_name* command line option, described in Chapter 30, **Running Tests from the Command Line**.

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**)

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 37, **Setting Testing Options from a Test Script**.

Maximum Length of List Item to Record

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 ListView 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])

Books Online

Find

Find Again

Help

Top of Chapter

Back

Run Tab

The Run tab options affect how WinRunner runs tests. You can set additional options for running tests using the Advanced Run Options dialog box, which you can open from the Run tab of the General Options dialog box. For information on the Advanced Run Options dialog box, see **Advanced Run Options Dialog Box** on page 877.

Books Online

Find

Find Again

Help

Top of Chapter

Back
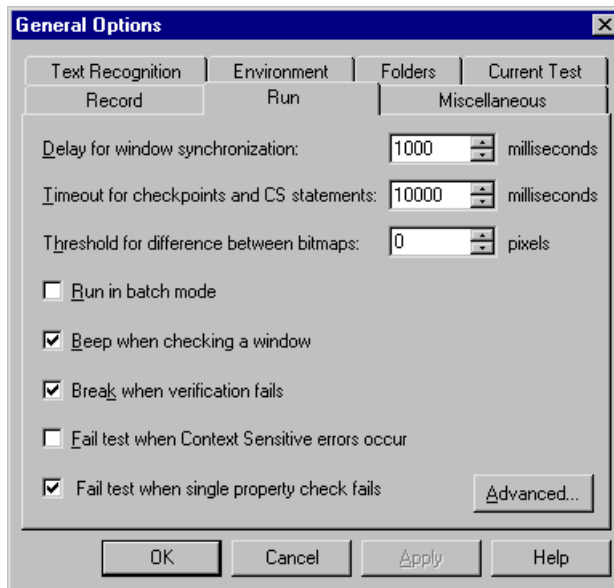
Delay for Window Synchronization

This option sets the sampling interval (in milliseconds) 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 2,000 milliseconds and the timeout is 10,000 milliseconds, 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 = **1000** [milliseconds])

Note: This option is accurate to within 20-30 milliseconds.

Note that you can use the **setvar** and **getvar** functions to set and retrieve the value of the corresponding *delay_msec* testing option from within a test script, as described in Chapter 37, **Setting Testing Options from a Test Script**.

Note that you can also set this option using the corresponding *-delay_msec* command line option, described in Chapter 30, **Running Tests from the Command Line**.

Books Online

Find

Find Again

Help

Top of Chapter

Back

Timeout for Checkpoints and CS Statements

This option sets the global timeout (in milliseconds) used by WinRunner when executing checkpoints and Context Sensitive statements. This value is added to the *time* parameter embedded in GUI checkpoint or synchronization point statements to determine the maximum amount of time that WinRunner searches for the specified window. The timeout must be greater than the delay for window synchronization (as set in the **Delay for Window Synchronization** box above).

(Default = **10000** [milliseconds])

Note: This option is accurate to within 20-30 milliseconds.

Note that you can use the **setvar** and **getvar** functions to set and retrieve the value of the corresponding *timeout_msec* testing option from within a test script, as described in Chapter 37, **Setting Testing Options from a Test Script**.

Note that you can also set this option using the corresponding *-timeout_msec* command line option, described in Chapter 30, **Running Tests from the Command Line**.

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 37, **Setting Testing Options from a Test Script**.

Note that you can also set this option using the corresponding *-min_diff* command line option, described in Chapter 30, **Running Tests from the Command Line**.

Books Online

Find

Find Again

Help

Top of Chapter

Back

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 folder, 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 29, **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 37, **Setting Testing Options from a Test Script**.

Note that you can also set this option using the corresponding *-batch* command line option, described in Chapter 30, **Running Tests from the Command Line**.

Beep when Checking a 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 37, **Setting Testing Options from a Test Script**.

Note that you can also set this option using the corresponding *-beep* command line option, described in Chapter 30, **Running Tests from the Command Line**.

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 37, **Setting Testing Options from a Test Script**.

Note that you can also set this option using the corresponding *-mismatch_break* command line option, described in Chapter 30, **Running Tests from the Command Line**.

Fail Test when Context Sensitive Errors Occur

This option determines whether WinRunner fails a test when Context Sensitive errors occur. A Context Sensitive error is the failure of a Context Sensitive statement during a test run. Context Sensitive errors often occur when WinRunner cannot identify a GUI object.

For example, a Context Sensitive error will occur if you run a test containing a **set_window** statement with the name of a non-existent window. Context Sensitive errors can also occur when window names are ambiguous. For information about Context Sensitive functions, refer to the *TSL Online Reference*.

☑ When this option is selected, WinRunner fails the test run if a Context Sensitive statement fails during a test.

☐ When this option is cleared, WinRunner does not fail the test run if a Context Sensitive statement fails during a test.

(Default = **cleared**)

Note that you can use the **setvar** and **getvar** functions to set and retrieve the value of the corresponding *cs_fail* testing option from within a test script, as described in Chapter 37, **Setting Testing Options from a Test Script**.

Note that you can also set this option using the corresponding *-cs_fail* command line option, described in Chapter 30, **Running Tests from the Command Line**.

Books Online

Find

Find Again

Help

Top of Chapter

Back

Fail Test when Single Property Check Fails

This option fails a test run when **_check_info** statements fail. It also writes an event to the Test Results window for these statements. (You can create _check_info statements using the **Create > GUI Checkpoint > For Single Property** command.)

☑ When this option is selected, WinRunner fails the test run if a **_check_info** statement fails during a test.

☐ When this option is cleared, WinRunner does not fail the test run if a **_check_info** statement fails during a test.

(Default = **cleared**)

For information about the **check_info** functions, refer to the *TSL Online Reference*.

Note that you can use the **setvar** and **getvar** functions to set and retrieve the value of the corresponding *single_prop_check_fail* testing option from within a test script, as described in Chapter 37, **Setting Testing Options from a Test Script**.

Note that you can also set this option using the corresponding *-single_prop_check_fail* command line option, described in Chapter 30, **Running Tests from the Command Line**.

Advanced Run Options Dialog Box

You set options in the Advanced Run Options dialog box the same way that you set options in the Run tab of the General Options dialog box.

To set advanced run options:

1 Choose **Settings > General Options**. The General Options dialog box opens.

2 Click the **Run** tab.

3 Click **Advanced**.

4 Set an option, as described in **Global Testing Options** on page 858.

5 Click **OK** to apply your change and close the dialog box.

The **Advanced Run Options** dialog box opens.
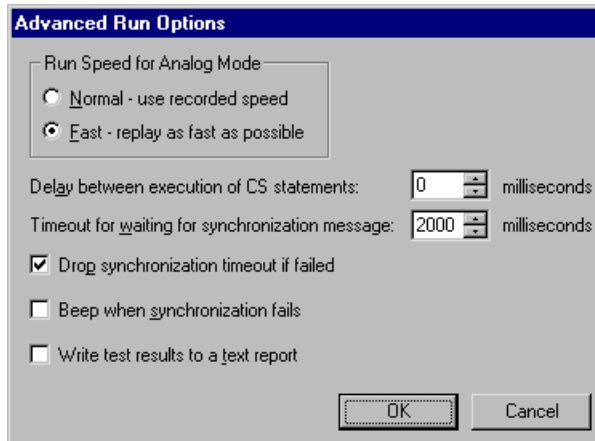
Run Speed for Analog Mode

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 37, **Setting Testing Options from a Test Script**.

Note that you can also set this option using the corresponding -*speed* command line option, described in Chapter 30, **Running Tests from the Command Line**.

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 37, **Setting Testing Options from a Test Script**.

Note that you can also set this option using the corresponding -*cs_run_delay* command line option, described in Chapter 30, **Running Tests from the Command Line**.

Books Online

M Find

Find Again

? Help

Top of Chapter

Back

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 37, **Setting Testing Options from a Test Script**.

Note: If synchronization often fails during your test runs, consider increasing the value of this option.

Books Online

M Find

Find Again

? Help

Top of Chapter

← Back

Drop Synchronization Timeout if Failed

This option determines whether WinRunner drops the synchronization timeout (as defined in the **Timeout for Waiting for Synchronization Message** option above) 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 37, **Setting Testing Options from a Test Script**.

Books Online

Find

Find Again

Help

Top of Chapter

Back

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 37, **Setting Testing Options from a Test Script**.

Write Test Results to a Text Report

This option instructs WinRunner to automatically write test results to a text report, called *report.txt*, which is saved in the results folder.

☑ 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 that you can also set this option using the corresponding -*create_text_report* command line option, described in Chapter 30, **Running Tests from the Command Line**.

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.
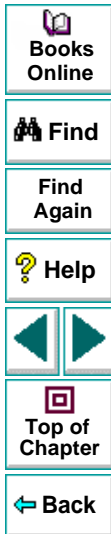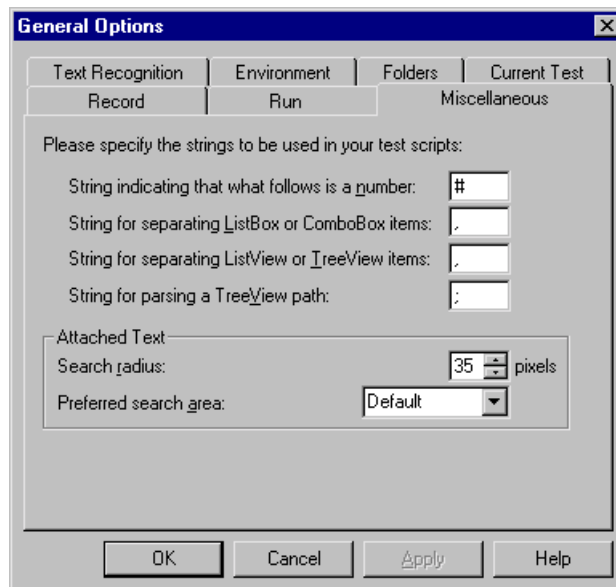
String Indicating that what Follows is a Number

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 item appearing 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 37, **Setting Testing Options from a Test Script**.

Books Online

Find

Find Again

Help

Top of Chapter

Back

String for Separating ListBox or ComboBox Items

This option defines the string recorded in the test script to separate items in a ListBox or a ComboBox.

(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 37, **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 ListView or a TreeView.

(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 37, **Setting Testing Options from a Test Script**.
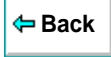
String for Parsing a TreeView Path

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 37, **Setting Testing Options from a Test Script**.

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 options: 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 values for the attached text options that you used when you recorded the test. Otherwise, WinRunner may not identify your GUI object.
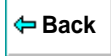
Attached Text - Search Radius

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 37, **Setting Testing Options from a Test Script**.
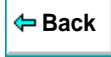
Attached Text - Preferred Search Area

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**)
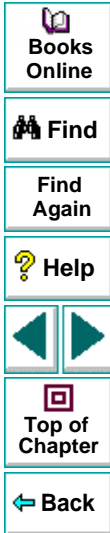
Note that you can use the **setvar** and **getvar** functions to set and retrieve the value of the corresponding *attached_text_area* testing option from within a test script, as described in Chapter 37, **Setting Testing Options from a Test Script**.

Books Online

Find

Find Again

Help

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.

Note: A text report of the test results can also be created from the Test Results window by choosing the Tools > Text Report command.

Top of Chapter

Back

Text Recognition Tab

The Text Recognition tab options affect how WinRunner recognizes text in your application.
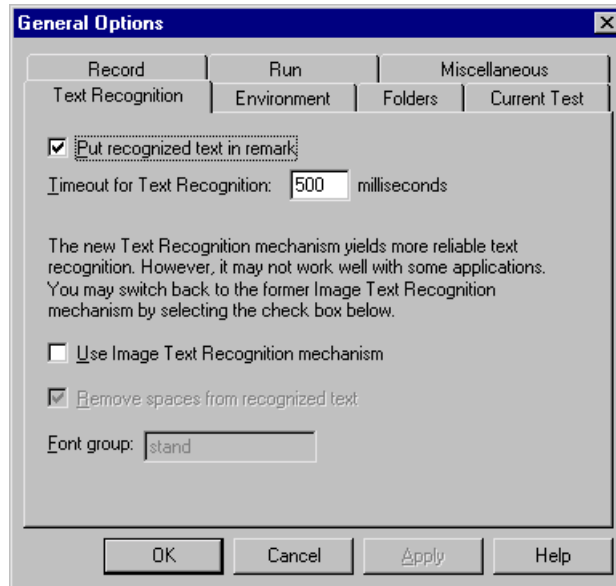
Put Recognized Text in Remark

When you create a text checkpoint, this option determines how WinRunner displays the captured text in the test 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 Create > Get Text > From Object/Window, 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**)

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 Mechanism

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**)

Remove Spaces from Recognized Text

This option removes multiple leading and trailing blanks in recognized text.

☑ If this option is selected, WinRunner removes multiple leading and trailing blank spaces found in recognized text during test creation from the test script.

☐ If this option is cleared, WinRunner transfers multiple 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 Text Recognition mechanism.

Font Group

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 428.

(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 37, **Setting Testing Options from a Test Script**.

Note that you can also set this option using the corresponding *-fontgrp* command line option, described in Chapter 30, **Running Tests from the Command Line**.

**Books
Online**

M Find

**Find
Again**

? Help

**Top of
Chapter**

⇐ Back

Environment Tab

The Environment tab options affect WinRunner's testing environment.
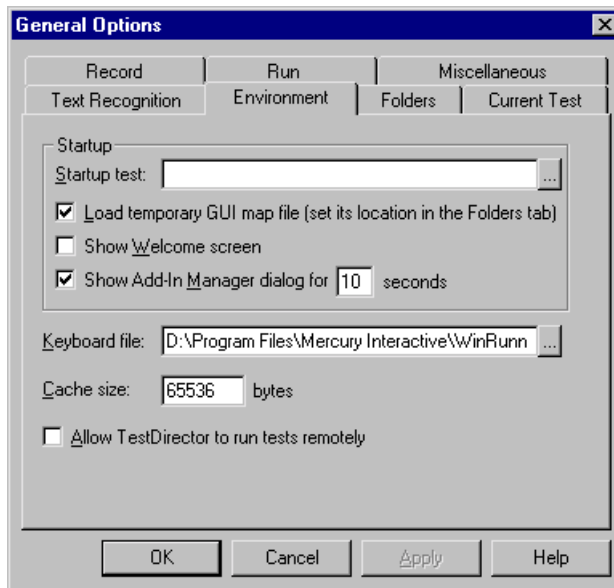
WinRunner Interface Language

If WinRunner is installed on a non-English operating system, you may have an option to select the WinRunner interface language from Environment tab of the General Options dialog box.

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 folder**)

Note: A startup test can be used in addition to (and not instead of) the initialization (tslinit) test.

Books Online

Find

Find Again

Help

Top of Chapter

Back

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 **General Options dialog box**. For more information, see **Temporary GUI Map File** on page 903.

Show 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**)

Display the Add-In Manager dialog

This option determines:

- whether to display the **Add-In Manager** dialog box when starting WinRunner

- if the **Add-In Manager** dialog box is displayed when starting WinRunner, how many seconds it remains open before it closes (timeout)
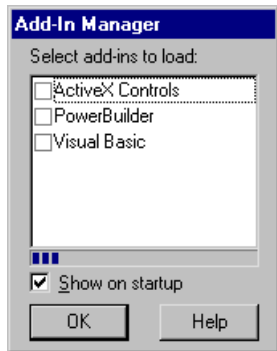
For information about the **Add-In Manager** dialog box and loading installed add-ins when starting WinRunner, see **Loading WinRunner Add-Ins** on page 50.

☑ If this option is selected, the **Add-In Manager** dialog box is displayed when starting WinRunner. In the timeout box, specify the number of seconds to wait before closing the window.

☐ If this option is cleared, the **Add-In Manager** dialog box is not displayed when starting WinRunner. The timeout box is disabled.

(Default = **selected** and **10** [seconds])

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 folder\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])

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 Agent application to your Windows startup. If the WinRunner Remote Agent application is not currently running on your machine, selecting this option starts it. When this application is running, the WinRunner Remote Agent 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 Agent application is added to your Windows startup. If the WinRunner Remote Agent 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 Agent 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.
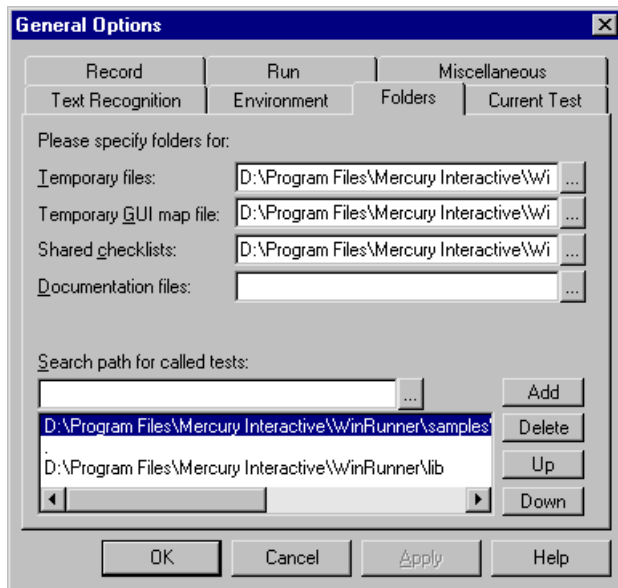
Temporary Files

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 folder\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 37, **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 General 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 folder\dat**)
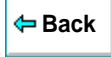
Shared Checklists

This box designates the folder in which WinRunner stores shared checklists for GUI and database checkpoints. In the test script, shared checklist files are designated by SHARED_CL before the file name in a **win_check_gui**, **obj_check_gui**, **check_db**, 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 Folder** on page 216. For more information on shared database checklists, see **Saving a Database Checklist in a Shared Folder** on page 372.

(Default = **installation folder\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 37, **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 folder\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 folder\lib**)

For more information, see Chapter 21, **Calling Tests**.
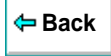
Note: When WinRunner is connected to TestDirector, you can specify the paths in a TestDirector database that WinRunner searches for called tests. Search paths in a TestDirector database can be preceded by [TD]. Note that you cannot use the Browse button to specify search paths in a TestDirector database: they must be typed directly into the search path box.
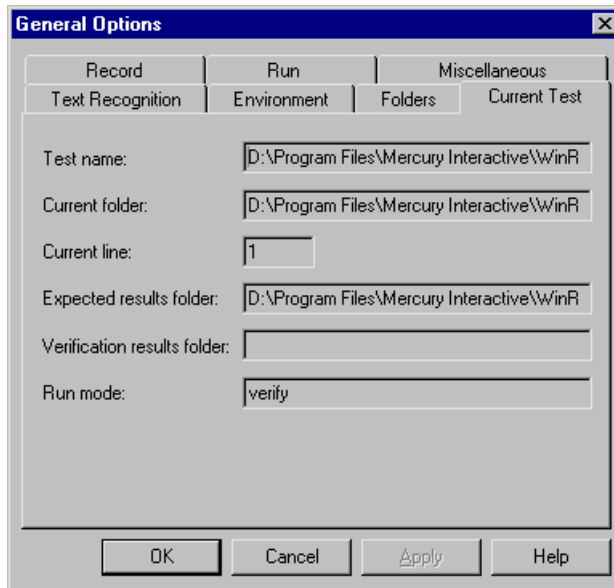
Note that you can use the **setvar** and **getvar** functions to set and retrieve the value of the corresponding *searchpath* testing option from within a test script, as described in Chapter 37, **Setting Testing Options from a Test Script**.

Note that you can also set this option using the corresponding *-search_path* command line option, described in Chapter 30, **Running Tests from the Command Line**.

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.

General Options ☒

Record	Run	Miscellaneous

Text Recognition	Environment	Folders	Current Test

Test name: D:\Program Files\Mercury Interactive\WinR

Current folder: D:\Program Files\Mercury Interactive\WinR

Current line: 1

Expected results folder: D:\Program Files\Mercury Interactive\WinR

Verification results folder:

Run mode: verify

OK	Cancel	Apply	Help

Test Name

This box 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 37, **Setting Testing Options from a Test Script**.

Current Folder

This box displays the current working folder 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 37, **Setting Testing Options from a Test Script**.

Current Line

This box 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 37, **Setting Testing Options from a Test Script**.

Books Online

Find

Find Again

Help

Top of Chapter

Back

Expected Results Folder

This box displays the full path of the expected results folder 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 37, **Setting Testing Options from a Test Script**.

Note that you can also set this option using the corresponding *-exp* command line option, described in Chapter 30, **Running Tests from the Command Line**.

Verification Results Folder

This box displays the full path of the verification results folder 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 37, **Setting Testing Options from a Test Script**.

Run Mode

This box 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 37, **Setting Testing Options from a Test Script**.

Books Online

Find

Find Again

Help

Top of Chapter

Back

Configuring WinRunner
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:

- **Setting Testing Options with setvar**
- **Retrieving Testing Options with getvar**
- **Controlling the Test Run with setvar and getvar**
- **Test Script Testing Options**

About Setting Testing Options from a Test Script

WinRunner testing options affect how you record test scripts and run tests. For example, you can set the speed at which WinRunner executes a test or determine how WinRunner records keyboard input.

You can set and retrieve the values of testing options from within a test script. To set the value of a testing option, use the **setvar** function. To retrieve the current value of a testing option, use the **getvar** function. By using a combination of **setvar** and **getvar** statements in a test script, you can control how WinRunner executes a test. You can use these functions to set and view the testing options for all tests, for a single test, or for part of a single test. You can also use these functions in a startup test script to set environment variables.

Most testing options can also be set using the General Options dialog box. For more information on setting testing options using the General Options dialog box, see Chapter 36, **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	single_prop_check_fail
beep	list_item_separator	speed
cs_run_delay	listview_item_separator	sync_fail_beep
cs_fail	min_diff	synchronization_timeout
delay_msec	mismatch_break	tempdir
drop_sync_timeout	rec_item_name	timeout_msec
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 with another **setvar** statement or from the corresponding **Break when verification fails** check box in the Run tab of the General Options dialog box.

Books Online

Find

Find Again

Help

Top of Chapter

Back

Using the **setvar** function changes a testing option globally, and this change is reflected in the General Options dialog box. However, you can also use the **setvar** function to set testing options for 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_msec* testing option to 20,000 for a specific test only, insert the following at the beginning of your test script:

```
# Keep the original value of the 'delay_msec' testing option
old_delay = getvar ("delay_msec") ;
setvar ("delay_msec", "20,000") ;
```

To change back the *delay* testing option to its original value at the end of the test, insert the following at the end of your test script:

```
#Change back the 'delay_msec' testing option to its original value.
setvar ("delay_msec", old_delay) ;
```

Books Online

M Find

Find Again

? Help

Top of Chapter

← Back

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	line_no	speed
attached_text_search_radius	list_item_separator	sync_fail_beep
batch	listview_item_separator	synchronization_timeout
beep	min_diff	td_log_dirname
cs_fail	mismatch_break	td_connection
cs_run_delay	rec_item_name	td_cycle_name
curr_dir	rec_owner_drawn	td_database_name
delay_msec	result	td_server_name
drop_sync_timeout	runmode	td_user_name
enum_descendent_toplevel	searchpath	tempdir
exp	shared_checklist_dir	testname
fontgrp	single_prop_check_fail	timeout_msec
item_number_seq	silent_mode	treeview_path_separator
key_editing		

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 General 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 the Test Run with setvar and getvar

You can use **getvar** and **setvar** together to control a test run without changing global settings. In the following test script fragment, WinRunner checks the bitmap Img1. The **getvar** function retrieves the values of the *timeout_msec* and *delay_msec* testing options, and **setvar** assigns their values for this **win_check_bitmap** statement. After the window is checked, **setvar** restores the values of the testing options.

t = getvar ("timeout_msec");
d = getvar ("delay_msec");
setvar ("timeout_msec", 30000);
setvar ("delay_msec", 3000);
win_check_bitmap ("calculator", Img1, 2, 261,269,93,42);
setvar ("timeout_msec", t);
setvar ("delay_msec", d);

Note: You can use the **setvar** and **getvar** functions in a startup test script to set environment variables for a specific WinRunner session.

Books Online

M Find

Find Again

? Help

Top of Chapter

Back

Test Script Testing Options

This section describes the WinRunner testing options that can be used with the **setvar** and **getvar** functions from within a test script. If you can also use the General Options dialog box to set or view an option, it is indicated below.

Books Online

M Find

Find Again

? Help

Top of Chapter

⬅ Back

attached_text_area

This option specifies the point on a GUI object from which WinRunner searches for its attached text.

Value	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

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 - Preferred search area** box in the Miscellaneous tab of the General Options dialog box, described in Chapter 36, **Setting Global Testing Options**.

Notes: When you run a test, you must use the same values for the attached text options that you used when you recorded the test. Otherwise, WinRunner may not identify the GUI object.

In previous versions of WinRunner, you could not set the preferred search area: WinRunner searched for attached text based on what is now the Default setting for the preferred search area. If backward compatibility is important, choose the Default setting.

Books Online

Find

Find Again

Help

Top of Chapter

Back

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 - Search radius** box in the Miscellaneous tab of the General Options dialog box, described in Chapter 36, **Setting Global Testing Options**.

Note: When you run a test, you must use the same values for the attached text options that you used when you recorded the test. Otherwise, WinRunner may not identify the GUI object.

---

batch

This option displays whether WinRunner is running in batch mode. In batch mode, WinRunner suppresses messages during a test run so that a test can run unattended. WinRunner also saves all the expected and actual results of a test run in batch mode in one folder, and displays them in one Test Results window. For more information on the batch testing option, see Chapter 29, **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 = **0**)

Note that you may also set this option using the **Run in batch mode** check box in the Run tab of the General Options dialog box, described in Chapter 36, **Setting Global Testing Options**.

Note that you can also set this option using the corresponding *-batch* command line option, described in Chapter 30, **Running Tests from the Command Line**.

Note: When you run tests in batch mode, you automatically run them in silent mode. For information about the *silent_mode* testing option, see **page 938**.

Books
Online

M Find

**Find
Again**

? Help

**Top of
Chapter**

← Back

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 a window** check box in the Run tab of the General Options dialog box, described in Chapter 36, **Setting Global Testing Options**.

Note that you can also set this option using the corresponding *-beep* command line option, described in Chapter 30, **Running Tests from the Command Line**.

cs_fail

This option determines whether WinRunner fails a test when Context Sensitive errors occur. A Context Sensitive error is the failure of a Context Sensitive statement during a test. Context Sensitive errors are often due to WinRunner's failure to identify a GUI object.

For example, a Context Sensitive error will occur if you run a test containing a **set_window** statement with the name of a non-existent window. Context Sensitive errors can also occur when window names are ambiguous. For information about Context Sensitive functions, refer to the *TSL Online Reference*.

You can use this option with the **setvar** and **getvar** functions.

(Default = **0**)

Note that you may also set this option using the corresponding **Fail test when Context Sensitive errors occur** check box in the Run tab of the General Options dialog box, described in Chapter 36, **Setting Global Testing Options**.

Note that you can also set this option using the corresponding *-cs_fail* command line option, described in Chapter 30, **Running Tests from the Command Line**.

cs_run_delay

This option sets the time (in milliseconds) that WinRunner waits between executing Context Sensitive statements when running a test.

You can use this option with the **setvar** and **getvar** functions.

(Default = **0** [milliseconds])

Note that you may also set this option using the corresponding **Delay between execution of CS statements** box in the Advanced Run Options dialog box, described in Chapter 36, **Setting Global Testing Options**.

Note that you can also set this option using the corresponding *-cs_run_delay* command line option, described in Chapter 30, **Running Tests from the Command Line**.

curr_dir

This option displays the current working folder for the test.

You can use this option with the **getvar** function.

This option has no default value.

Note that you may also view the location of the current working folder for the test from the corresponding **Current folder** box in the Current Test tab of the General Options dialog box, described in Chapter 36, **Setting Global Testing Options**.

delay_msec

This option sets the sampling interval (in seconds) used to determine that a window is stable before capturing it for a Context Sensitive checkpoint or synchronization point. To be declared stable, a window must not change between two consecutive samplings. This sampling continues until the window is stable or the timeout (as set with the *timeout_msec* testing option) is reached. (Formerly *delay*, which was measured in seconds.)

For example, when the delay is two seconds and the timeout is ten seconds, WinRunner checks the window in the application under test every two seconds until two consecutive checks produce the same results or until ten seconds have elapsed. Setting the value to 0 disables all bitmap checking.

You can use this option with the **setvar** and **getvar** functions.

(Default = **1000** [milliseconds])

Note: This option is accurate to within 20-30 milliseconds.

Note that you may also set this option using the corresponding **Delay for window synchronization** box in the Run tab of the General Options dialog box, described in Chapter 36, **Setting Global Testing Options**.

Note that you can also set this option using the corresponding *-delay_msec* command line option, described in Chapter 30, **Running Tests from the Command Line**.

Books Online

M Find

Find Again

? Help

Top of Chapter

⇦ Back

drop_sync_timeout

This option determines whether WinRunner drops the synchronization timeout (as defined by the *synchronization_timeout* testing option) if synchronization fails.

(Default = **off**)

You can use this option with the **getvar** and **setvar** functions.

Note that you may also set this option using the corresponding **Drop synchronization timeout if failed** check box in the Advanced Run Options dialog box, described in Chapter 36, **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 set this option using the corresponding **Consider child windows** check box in the Record tab of the General Options dialog box, described in Chapter 36, **Setting Global Testing Options**.

Books Online

Find

Find Again

Help

Top of Chapter

Back

exp

This option displays the full path of the expected results folder associated with the current test run.

You can use this option with the **getvar** function.

This option has no default value.

Note that you may also view the full path of the expected results folder from the corresponding **Expected results folder** box in the Current Test tab of the General Options dialog box, described in Chapter 36, **Setting Global Testing Options**.

Note that you can also set this option using the corresponding -*exp* command line option, described in Chapter 30, **Running Tests from the Command Line**.

fontgrp

To be able to use Image Text Recognition (instead of the default Text Recognition), (described in **Use Image Text Recognition Mechanism** on page 893), 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 428.

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 General Options dialog box, described in Chapter 36, **Setting Global Testing Options**.

Note that you can also set this option using the corresponding *-fontgrp* command line option, described in Chapter 30, **Running Tests from the Command Line**.

Books Online

Find

Find Again

Help

Top of Chapter

Back

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 General Options dialog box, described in Chapter 36, **Setting Global Testing Options**.

Books Online

Find

Find Again

Help

Top of Chapter

Back

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, more readable type statements** check box in the Record tab of the General Options dialog box, described in Chapter 36, **Setting Global Testing Options**.

Books Online

M Find

Find Again

? Help

Top of Chapter

⇐ Back

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 the current line number in the test script from the corresponding **Current Line** box in the Current Test tab of the General Options dialog box, described in Chapter 36, **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 General Options dialog box, described in Chapter 36, **Setting Global Testing Options**.

Books Online

Find

Find Again

Help

Top of Chapter

Back

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 General Options dialog box, described in Chapter 36, **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 General Options dialog box, described in Chapter 36, **Setting Global Testing Options**.

Note that you can also set this option using the corresponding *-min_diff* command line option, described in Chapter 30, **Running Tests from the Command Line**.

mismatch_break

This option determines whether WinRunner pauses the test run and displays a message whenever verification fails or whenever any message is generated as a result of a context sensitive statement during a test that is run in Verify mode. This option should be used only when working interactively.

For example, if a **set_window** statement is missing from a test script, WinRunner cannot find the specified window. If this option is on, WinRunner pauses the test and opens the Run wizard to enable the user to locate the window. If this option is off, WinRunner reports an error in the Test Results window and proceeds to run the next statement in the test script.

You can use this option with the **setvar** and **getvar** functions.

(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 General Options dialog box, described in Chapter 36, **Setting Global Testing Options**.

Note that you can also set this option using the corresponding *-mismatch_break* command line option, described in Chapter 30, **Running Tests from the Command Line**.

Books Online

Find

Find Again

Help

Top of Chapter

Back

rec_item_name

This option determines whether WinRunner records non-unique ListBox and ComboBox items by name or by index.

You can use this option with the **setvar** and **getvar** functions.

(Default = **0**)

Note that you may also set this option using the corresponding **Record non-unique list items by name** check box in the Record tab of the General Options dialog box, described in Chapter 36, **Setting Global Testing Options**.

Note that you can also set this option using the corresponding *-rec_item_name* command line option, described in Chapter 30, **Running Tests from the Command Line**.

Books Online

M Find

Find Again

? Help

Top of Chapter

← Back

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 set this option using the corresponding **Record owner-drawn buttons** box in the Record tab of the General Options dialog box, described in Chapter 36, **Setting Global Testing Options**.

result

This option displays the full path of the verification results folder associated with the current test run.

You can use this option with the **getvar** function.

This option has no default value.

Note that you may also view the full path of the verification results folder from the corresponding **Verification results folder** box in the Current Test tab of the General Options dialog box, described in Chapter 36, **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.

This option has no default value.

Note that you may also view the current run mode from the corresponding **Run mode** box in the Current Test tab of the General Options dialog box, described in Chapter 36, **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 folder\lib**)

Note that you may also set this option using the corresponding **Search path for called tests** box in the Folders tab of the General Options dialog box, described in Chapter 36, **Setting Global Testing Options**.

Note that you can also set this option using the corresponding -*search_path* command line option, described in Chapter 30, **Running Tests from the Command Line**.

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 option designates the folder in which WinRunner stores shared checklists for GUI and database checkpoints. In the test script, shared checklist files are designated by SHARED_CL before the file name in a **win_check_gui**, **obj_check_gui**, **check_gui,** or **check_db** statement. For more information on shared GUI checklists, see **Saving a GUI Checklist in a Shared Folder** on page 216. For more information on shared database checklists, see **Saving a Database Checklist in a Shared Folder** on page 372. 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 folder\chklist**)

Note that you may also view the location of the folder in which WinRunner stores shared checklists from the corresponding **Shared checklists** box in the Folders tab of the General Options dialog box, described in Chapter 36, **Setting Global Testing Options**.

Books Online

Find

Find Again

Help

silent_mode

This option displays whether WinRunner is running in silent mode. In silent mode, WinRunner suppresses messages during a test run so that a test can run unattended. When you run a test remotely from TestDirector, you must run it in silent mode, because no one is monitoring the computer where the test is running to view the messages. For information on running tests remotely from TestDirector, see Chapter 40, **Managing the Testing Process**.

You can use this option with the **getvar** function.

(Default = **off**)

Top of Chapter

Back

Note: When you run tests in batch mode, you automatically run them in silent mode. For information running tests in batch mode, see Chapter 29, **Running Batch Tests**.

single_prop_check_fail

This option fails a test run when **_check_info** statements fail. It also writes an event to the Test Results window for these statements. (You can create **_check_info** statements using the **Create > GUI Checkpoint > For Single Property** command.)

You can use this option with the **setvar** and **getvar** functions.

(Default = **1**)

For information about the **check_info** functions, refer to the *TSL Online Reference*.

Note that you may also set this option using the corresponding **Fail test when single property check fails** option in the Run tab of the General Options dialog box, described in Chapter 36, **Setting Global Testing Options**.

Note that you can also set this option using the corresponding *-single_prop_check_fail* command line option, described in Chapter 30, **Running Tests from the Command Line**.

Books Online

Find

Find Again

Help

Top of Chapter

Back

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 Advanced Run Options dialog box, described in Chapter 36, **Setting Global Testing Options**.

Note that you can also set this option using the corresponding -*speed* command line option, described in Chapter 30, **Running Tests from the Command Line**.

Books Online

Find

Find Again

? Help

Top of Chapter

Back

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 Advanced Run Options dialog box, described in Chapter 36, **Setting Global Testing Options**.

Note: You 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 (described below) or the corresponding **Timeout for waiting for synchronization message** option in the Advanced Run 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 Advanced Run Options dialog box, described in Chapter 36, **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.

(Default = **off**)

Note that you can connect to TestDirector from the **Connection to TestDirector** dialog box, which you open by choosing Tools > TestDirector Connection. For more information about connecting to TestDirector, see Chapter 40, **Managing the Testing Process**.

Note that you can also set this option using the corresponding *-td_connection* command line option, described in Chapter 30, **Running Tests from the Command Line**.

Books Online

M Find

Find Again

? Help

Top of Chapter

Back

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 1008. You may also set this option from within TestDirector. For more information, refer to the *TestDirector User's Guide*.

Note that you can also set this option using the corresponding *-td_cycle_name* command line option, described in Chapter 30, **Running Tests from the Command Line**.

td_database_name

This option displays the name of the TestDirector project 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 **Project Connection** box in the **Connection to TestDirector** dialog box, which you can open by choosing Tools > TestDirector Connection. For more information, see Chapter 40, **Managing the Testing Process**.

Note that you can also set this option using the corresponding *-td_database_name* command line option, described in Chapter 30, **Running Tests from the Command Line**.

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 **Server Connection** box in the **Connection to TestDirector** dialog box, which you can open by choosing Tools > TestDirector Connection. For more information, see Chapter 40, **Managing the Testing Process**.

Note that you can also set this option using the corresponding *-td_server_name* command line option, described in Chapter 30, **Running Tests from the Command Line**.

Books Online

Find

Find Again

Help

Top of Chapter

Back

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 can also set this option using the corresponding *-td_user_name* command line option, described in Chapter 30, **Running Tests from the Command Line**.

Note that you may set this option using the **User Name** box in the **Connection to TestDirector** dialog box, which you can open by choosing Tools > TestDirector Connection. For more information, see Chapter 40, **Managing the Testing Process**.

Books Online

M Find

Find Again

? Help

Top of Chapter

Back

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 folder\tmp**)

Note that you may also set this option using the corresponding **Temporary files** box in the Folders tab of the General Options dialog box, described in Chapter 36, **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 view the full path of the current test from the corresponding **Test Name** box in the Current Test tab of the General Options dialog box, described in Chapter 36, **Setting Global Testing Options**.

Books Online

M Find

Find Again

? Help

Top of Chapter

Back

timeout_msec

This option sets the global timeout (in milliseconds) used by WinRunner when executing checkpoints and Context Sensitive statements. This value is added to the *time* parameter embedded in GUI checkpoint or synchronization point statements to determine the maximum amount of time that WinRunner searches for the specified window. The timeout must be greater than the delay for window synchronization (as set with the *delay_msec* testing option). (Formerly *timeout*, which was measured in seconds.)

For example, in the statement:

win_check_bitmap ("calculator", Img1, 2, 261,269,93,42);

when the *timeout_msec* variable is 10,000 milliseconds, this operation takes a maximum of 12,000 (2,000 +10,000) seconds.

You can use this option with the **setvar** and **getvar** functions.

(Default = **10000** [milliseconds])

Note: This option is accurate to within 20-30 milliseconds.

Books
Online

M Find

Find
Again

? **Help**

Top of
Chapter

Back

Note that you may also set this option using the corresponding **Timeout for checkpoints and CS statements** box in the Run tab of the General Options dialog box, described in Chapter 36, **Setting Global Testing Options**.

Note that you can also set this option using the corresponding *-timeout_msec* command line option, described in Chapter 30, **Running Tests from the Command Line**.

treeview_path_separator

This option defines the string recorded in the test script to separate items in a tree view path.

You can use this option with the **getvar** and **setvar** functions.

(Default = **;**)

Note that you may also set this option using the corresponding **String for parsing a TreeView path** box in the Miscellaneous tab of the General Options dialog box, described in Chapter 36, **Setting Global Testing Options**.

Books Online

Find

Find Again

? Help

Top of Chapter

Back

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:

- **Adding a Category to the Function Generator**
- **Adding a Function to the Function Generator**
- **Associating a Function with a Category**
- **Adding a Subcategory to a Category**
- **Setting a Default Function for a Category**

Books Online

Find

Find Again

Help

Top of Chapter

Back

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 > For 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 Function Generator**, click the **Insert Function from Function Generator** button on the User toolbar, or press the INSERT FUNCTION FROM FUNCTION GENERATOR softkey.)

2 In the **Category** box, click **function table**.

3 In the **Function Name** box, click **generator_add_category**.

4 Click **Args**. The Function Generator expands.

5 In the **Category Name** box, type the name of the new category between the quotes. Click **Paste** to paste the TSL statement into your test script.

6 Click **Close** to close the Function Generator.

A **generator_add_category** statement is inserted into your test script.

Note: You must run the test script in order to insert a new category into the Function Generator.

Adding a Function to the Function Generator

When you add a function to the Function Generator, you specify the following:

- how the user supplies values for the arguments in the function
- the function description that appears in the Function Generator

Note that after you add a function to the Function Generator, you should associate the function with a category. See **Associating a Function with a Category** on page 967.

You use the **generator_add_function TSL function** to add a user-defined function to the Function Generator.

Books Online

🔍 Find

Find Again

❓ Help

Top of Chapter

⬅ Back

To add a function to the Function Generator:

 1 Open the **Function Generator**. (Choose **Create > Insert Function > From Function Generator**, click the **Insert Function from Function Generator** button on the User toolbar, or press the INSERT FUNCTION FROM FUNCTION GENERATOR softkey.)

 2 In the **Category** box, click **function table**.

 3 In the **Function Name** box, click **generator_add_function**.

 4 Click **Args**. The Function Generator expands.

 5 In the Function Generator, define the *function_name*, *description*, and *arg_number* arguments:

   - In the *function_name* box, type the name of the new function between the quotes. Note that you can include spaces and upper-case letters in the function name.

   - In the *description* box, enter the description of the function between the quotes. Note that it does not have to be a valid string expression and it must not exceed 180 characters.

   - In the *arg_number* box, you must choose 1. To define additional arguments (up to eight arguments for each new function), you must manually modify the generated **generator_add_function** statement once it is added to your test script.
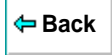
6 For the function's first argument, define the following arguments: *arg_name*, *arg_type*, and *default_value* (if relevant):

- In the *arg_name* box, type the name of the argument within the quotation marks. Note that you can include spaces and upper-case letters in the argument name.

- In the *arg_type* box, select **browse**, **point_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 958.

- In the *default_value* box, if relevant, choose the default value for the argument.

- Note that any additional arguments for the new function cannot be added from the Function Generator: The *arg_name*, *arg_type*, and *default_value* arguments must be added manually to the **generator_add_function** statement in your test script.

7 Click **Paste** to paste the TSL statement into your test script.

8 Click **Close** to close the Function Generator.

Note: You must run the test script in order to insert a new function into the Function Generator.

Defining Function Arguments

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

generator_add_function (*function_name, description, arg_number,*
arg_name_1, arg_type_1, default_value_1,
               ...
arg_name_n, arg_type_n, default_value_n **);**

- *function_name* is the name of the function you are adding.

- *description* is a brief explanation of the function. The description appears in the Description box of the Function Generator when the function is selected. It does not have to be a valid string expression and must not exceed 180 characters.

- *arg_number* is the number of arguments in the function. This can be any number from zero to eight.

  For each argument in the function you define, you supply the name of the argument, how it is filled in, and its default value (where possible). When you define a new function, you repeat the following parameters for each argument in the function: *arg_name, arg_type,* and *default_value.*

- *arg_name* defines the name of the argument that appears in the Function Generator.

- *arg_type* defines how the user fills in the argument's value in the Function Generator. There are five types of arguments.

browse():   The value of the argument is evaluated by pointing to a file in a browse file dialog box. Use *browse* when the argument is a file. To select a file with specific file extensions only, specify a list of default extension(s). Items in the list should be separated by a space or tab. Once a new function is defined, the *browse* argument is defined in the Function Generator by using a Browse button.

point_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(0 1):   The value of the argument is selected from a list. Use *select_list* when there is a limited number of argument values, and you can supply all the values. Once a new function is defined, the *select_list* argument is defined in the Function Generator by using a combo box.

type_edit: The value of the argument is typed in. Use *type_edit* when you cannot supply the full range of argument values. Once a new function is defined, the *type_edit* argument is defined in the Function Generator by typing into an edit field.
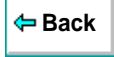
- *default_value* provides the argument's default value. You may assign default values to **select_list** and **type_edit** arguments. The default value you specify for a **select_list** argument must be one of the values included in the list. You cannot assign default values to **point_window** and **point_object** arguments.

  The following are examples of argument definitions that you can include in **generator_add_function** statements. The examples include the syntax of the argument definitions, their representations in the Function Generator, and a brief description of each definition.

Example 1

generator_add_function ("window_name","This function...",1,
   "Window Name","point_window","");

The *function_name* is window_name. The *description* is "This function...". The *arg_number* is 1. The *arg_name* is Window Name. The *arg_type* is point_window. There is no *default_value*: since the argument is selected by pointing to a window, this argument is an empty string.

When you select the "window_name" function in the Function Generator and click the Args button, the Function Generator appears as follows:
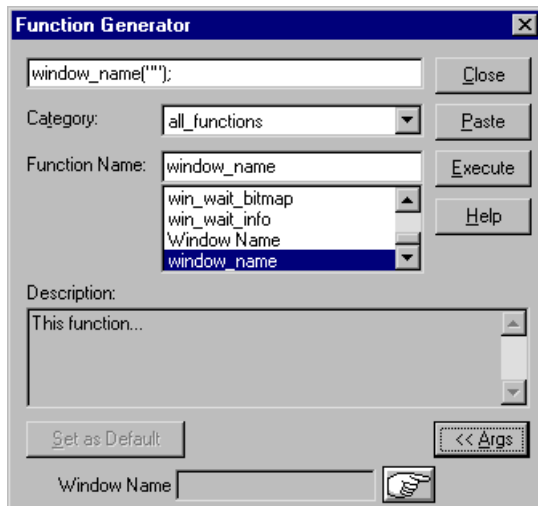
Example 2

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:
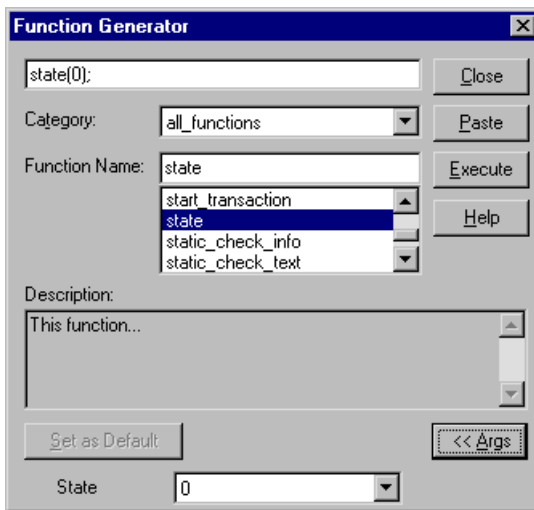
Example 3

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:

Books Online

M Find

Find Again

? Help

Top of Chapter

← Back
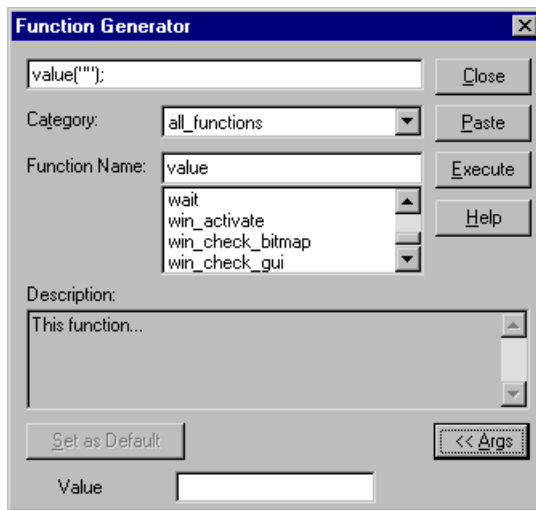
Defining Property Arguments

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.

Books Online

M Find

Find Again

? Help

Top of Chapter

Back
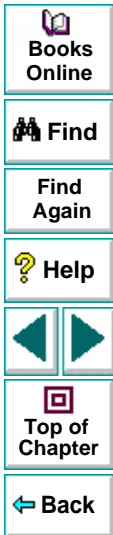
Example 1

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)");
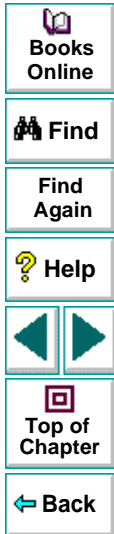
The first three arguments in **generator_add_function** define the following:

● 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**.

Books Online

Find

Find Again

Help

Top of Chapter

Back

To associate a function with a category:

1 Open the **Function Generator**. (Choose **Create > Insert Function > From Function Generator**, click the **Insert Function from Function Generator** button on the User toolbar, or press the INSERT FUNCTION FROM FUNCTION GENERATOR softkey.)

2 In the **Category** box, click **function table**.

3 In the **Function Name** box, click **generator_add_function_to_category**.

4 Click **Args**. The Function Generator expands.

5 In the **Category Name** box, enter the category name as it already appears in the Function Generator.

6 In the **Function Name** box, enter the function name as it already appears in the Function Generator.

7 Click **Paste** to paste the TSL statement into your test script.

8 Click **Close** to close the Function Generator.

A **generator_add_function_to_category** statement is inserted into your test script. In the following example, the "check_my_button_label" function is associated with the "my_button" category. This example assumes that you have already added the "my_button" category and the "check_my_button_label" function to the Function Generator.

generator_add_function_to_category ("my_button", "check_my_button_label");

Note: You must run the test script in order to associate a function with a category.

Books Online

M Find

Find Again

? Help

Top of Chapter

Back

Adding a Subcategory to a Category

You use the **generator_add_subcategory** TSL function to make one category a subcategory of another category. Both categories must already exist. The **generator_add_subcategory** function adds all the functions in the subcategory to the list of functions for the parent category.

If you create a separate category for your new functions, you can use the **generator_add_subcategory** function to add the new category as a subcategory of the relevant Context Sensitive category.

The syntax of **generator_add_subcategory** is as follows:

generator_add_subcategory (*category_name***,** *subcategory_name* **);**

- *category_name* is the name of an existing category in the Function Generator.

- *subcategory_name* is the name of an existing category in the Function Generator.

To add a subcategory to a category:

1 Open the **Function Generator**. (Choose **Create > Insert Function > From Function Generator**, click the **Insert Function from Function Generator** button on the User toolbar, or press the INSERT FUNCTION FROM FUNCTION GENERATOR softkey.)

2 In the **Category** box, click **function table**.

3 In the **Function Name** box, click **generator_add_subcategory**.

4 Click **Args**. The Function Generator expands.

5 In the **Category Name** box, enter the category name as it already appears in the Function Generator.

6 In the **Subcategory Name** box, enter the subcategory name as it already appears in the Function Generator.

7 Click **Paste** to paste the TSL statement into your test script.

8 Click **Close** to close the Function Generator.

A **generator_add_subcategory** statement is inserted into your test script. In the following example, the "my_button" category is defined as a subcategory of the "push_button" category. All "my_button" functions are added to the list of functions defined for the push_button category.

generator_add_subcategory ("push_button", "my_button");

Note: You must run the test script in order to add a subcategory to a category.

Setting a Default Function for a Category

You set the default function for a category using the **generator_set_default_function** TSL function. This function has the following syntax:

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

- *category_name* is an existing category.

- *function_name* is an existing function.

You can set a default function for a standard category or for a user-defined category that you defined using the **generator_add_category** function. If you do not define a default function for a user-defined category, WinRunner uses the first function in the list as the default function.

Note that the **generator_set_default_function** function performs the same operation as the Set As Default button in the Function Generator dialog box. However, a default function set through the Set As Default checkbox remains in effect during the current WinRunner session only. By adding **generator_set_default_function** statements to your startup test, you can set default functions permanently.

To add a subcategory to a category:

1 Open the **Function Generator**. (Choose **Create > Insert Function > From Function Generator**, click the **Insert Function from Function Generator** button on the User toolbar, or press the INSERT FUNCTION FROM FUNCTION GENERATOR softkey.)

2 In the **Category** box, click **function table**.

3 In the **Function Name** box, click **generator_set_default_function**.

4 Click **Args**. The Function Generator expands.

5 In the **Category Name** box, enter the category name as it already appears in the Function Generator.

6 In the **Default** box, enter the function name as it already appears in the Function Generator.

7 Click **Paste** to paste the TSL statement into your test script.

8 Click **Close** to close the Function Generator.

A **generator_set_default_function** statement is inserted into your test script. In the following example, the default function of the push button category is changed from **button_check_enabled** to the user-defined "check_my_button_label" function.

generator_set_default_function ("push_button", "check_my_button_label");

Note: You must run the test script in order to set a default function for a category.

Configuring
Initializing Special Configurations

By creating *startup tests*, you can automatically initialize special testing configurations each time you start WinRunner.

This chapter describes:

● **Creating Startup Tests**

● **Sample Startup Test**

About Initializing Special Configurations

A startup test is a test script that is automatically run each time you start WinRunner. You can create startup tests that load GUI map files and compiled modules, configure recording, and start the application under test.

You designate a test as a startup test by entering its location in the Startup Test box in the Environment tab in the General Options dialog box. For more information on using the General Options dialog box, see Chapter 36, **Setting Global Testing Options**.

Books Online

Find

Find Again

Help

Top of Chapter

Back

Creating Startup Tests

You should add the following types of statements to your startup test:

- **load** statements, which load compiled modules containing user-defined functions that you frequently call from your test scripts.

- **GUI_load** statements, which load one or more GUI map files. This ensures that WinRunner recognizes the GUI objects in your application when you run tests.

- statements that configure how WinRunner records GUI objects in your application, such as **set_record_attr** or **set_class_map.**

- an **invoke_application** statement, which starts the application being tested.

- statements that enable WinRunner to generate custom record TSL functions when you perform operations on custom objects, such as **add_cust_record_class**.

By including the above elements in a startup test, WinRunner automatically compiles all designated functions, loads all necessary GUI map files, configures the recording of GUI objects, and loads the application being tested.

Note 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

Books
Online

Find

Find
Again

Help

Top of
Chapter

Back
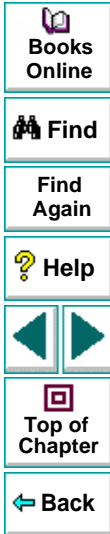
Working with TestSuite
Managing the Testing Process

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:

- **About Managing the Testing Process**

- **Using WinRunner with TestDirector**

- **Connecting to and Disconnecting from a Project**

- **Saving Tests to a Project**

- **Opening Tests in a Project**

- **Opening Tests in a Project**

- **Managing Test Versions in WinRunner**

- **Saving GUI Map Files to a Project**

- **Opening GUI Map Files in a Project**

- **Running Tests in a Test Set**

- **Running Tests on Remote Hosts**

- **Viewing Test Results from a Project**

- **Using TSL Functions with TestDirector**

- **Command Line Options for Working with TestDirector**

Books Online

Find

Find Again

? Help

Top of Chapter

Back

About Managing the Testing Process

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.

Books Online

Find

Find Again

Help

Top of Chapter

Back

TestDirector divides testing into three modes of operation: Plan Tests, Run Tests, and Track Defects. In Plan Tests 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.
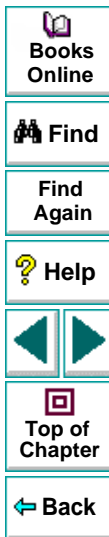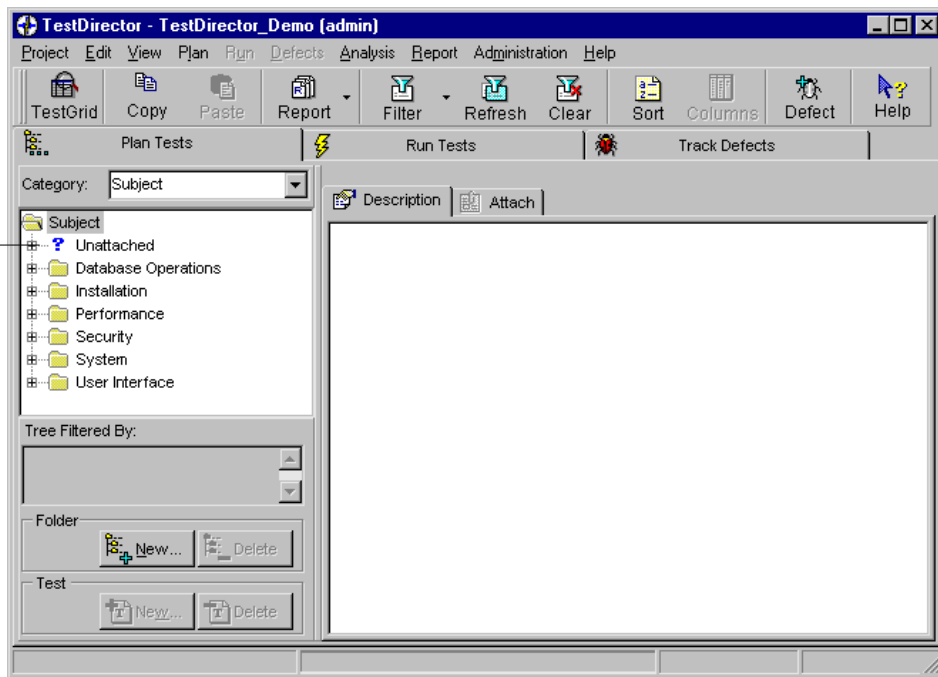
Test plan tree —

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 Run Tests 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 TestDirector test repository. Once you build a test set, you can schedule test runs. If your test set contains automated tests, TestDirector automatically opens WinRunner and runs 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 TestDirector runs a test in WinRunner, it displays the results and marks the test as passed, failed, or not completed.

Books Online

Find

Find Again

Help

Top of Chapter

Back

Test Director's Version Manager lets you update and revise your automated test scripts while maintaining old versions of each test. This helps you keep track of the changes made to each test script, see what was modified from one version of a script to another, or return to a previous version of the test script.

In Track Defects 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. After a test has been run, the results are viewed and analyzed in TestDirector.

TestDirector stores test and defect information in a project. TestDirector projects 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 projects within TestDirector. These projects store information related to the current testing project, such as tests, test run results, and reported defects.

In order for WinRunner to access the project, you must connect it to the TestDirector server. This is a program that handles the communication between WinRunner and the TestDirector project. Note that the TestDirector server usually runs on your TestDirector machine but you can also install it on any computer connected to the network.

WinRunner   *TestDirector Server*   *TestDirector Project*
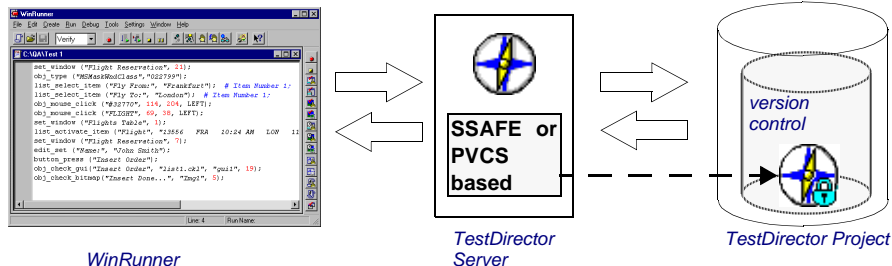
When WinRunner is connected to TestDirector, you can save a test by associating it with a subject in the test plan tree, instead of assigning the test to a folder in the file system. This makes it easy to organize tests by subject for your application. When you open a test, you search for it according to its position in the test plan tree. You can check test versions in and out of the Version Manager's version control database directly from WinRunner. When you run tests, results are sent directly to your TestDirector project.
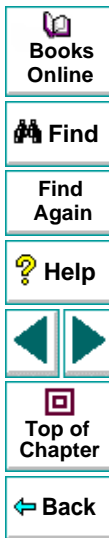
Note: The integration of WinRunner 6.0 with TestDirector is valid only for TestDirector 5.0 and higher. TestDirector with version control support is available only from TestDirector 6.0

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 General Options dialog box (Settings > General Options). For more information on setting this option, see Chapter 36, **Setting Global Testing Options**.

In Windows 95, 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, then WinRunner starts it, and the Remote Server icon appears in the status area of your screen.

Connecting to and Disconnecting from a Project

If you are working with both WinRunner and TestDirector, WinRunner can communicate with your TestDirector project. You can connect or disconnect WinRunner from a TestDirector project 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 server. This server handles the connections between WinRunner and the TestDirector project. Next, you choose the project you want WinRunner to access. The project stores tests and test run information for the application you are testing. Note that TestDirector projects are password protected, so you must provide a user name and a password.

Connecting WinRunner to a TestDirector Server and a Project

You must connect WinRunner to the TestDirector API server before you connect WinRunner to a project. For more information, see **Using WinRunner with TestDirector** on page 983.

To connect WinRunner to a TestDirector server and a project:

1 Choose **Tools > TestDirector Connection**.

The **Connection to TestDirector** dialog box opens.
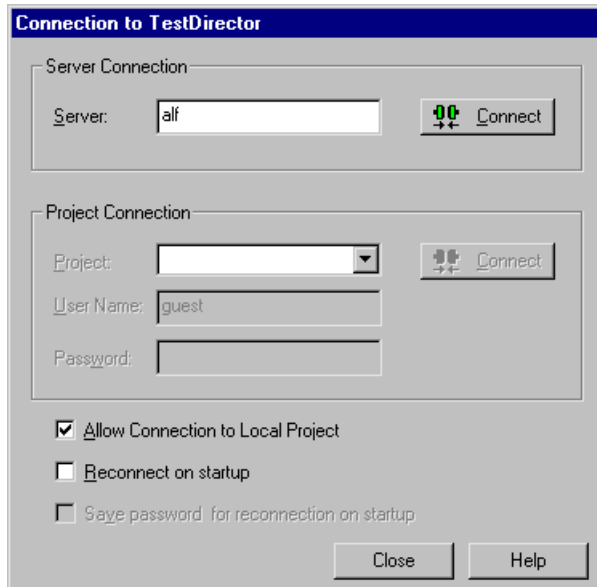
2 In the **Server Connection** section, in the **Server** box, enter the name of the host where the TestDirector server runs.

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 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.

Once the connection to the selected project is established, the project's name is displayed in read-only format in the **Project** box.

To automatically reconnect to the TestDirector server and the selected project on startup, select the **Reconnect on startup** check box.

If the **Reconnect on startup** check box is selected, then the **Save password for reconnection on startup** check box is enabled. To save your password for reconnection on startup, select the **Save password for reconnection on startup** check box. If you do not save your password, you will be prompted to enter it when WinRunner connects to TestDirector on startup.

8 Click **Close** to close the Connection to TestDirector dialog box.

Note: You can also connect WinRunner to a TestDirector server and project using the corresponding *-td_connection*, *-td_database_name*, *-td_password*, *-td_server_name*, *-td_user_name* command line options, described in Chapter 30, "Running Tests from the Command Line."

Books Online

Find

Find Again

Help

Top of Chapter

Back

Disconnecting from a TestDirector Project

You can disconnect from a TestDirector project. This enables you to select a different project while using the same TestDirector server.

To disconnect WinRunner from a project:

1 Choose **Tools > TestDirector Connection**.

The **Connection to TestDirector** dialog box opens.

Connection to TestDirector

Server Connection

Server: alf   ⬛ Disconnect

Project Connection

Project: TD

User Name: guest

Password:

☑ Allow Connection to Local Project

☐ Reconnect on startup

☐ Save password for reconnection on startup

Close   Help
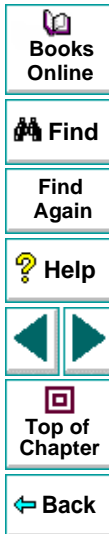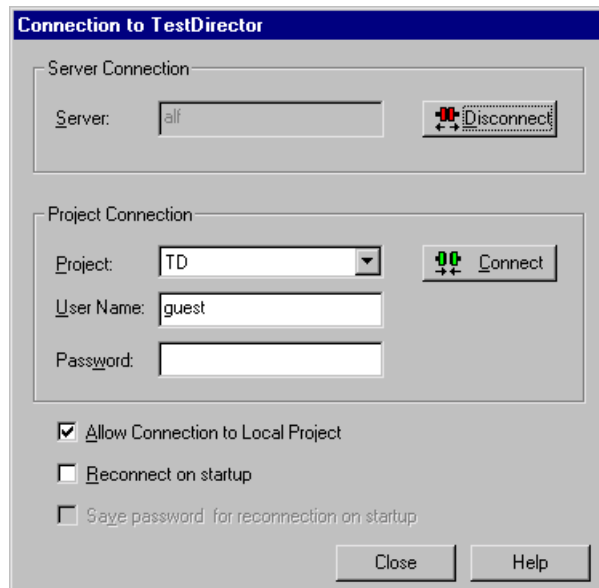
2 In the **Project Connection** section, click **Disconnect** to disconnect WinRunner from the selected project.

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.

To disconnect WinRunner from a server:

1 Choose **Tools > TestDirector Connection**.

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 that you can also disconnect WinRunner from a TestDirector server using the corresponding *-td_connection* and *-td_database_name* command line options, described in Chapter 30, "Running Tests from the Command Line."

Note: If you disconnect WinRunner from a TestDirector server without first disconnecting from a project, WinRunner's connection to that database is automatically disconnected.

Books Online

Find

Find Again

Help

Top of Chapter

Back

Saving Tests to a Project

When WinRunner is connected to a TestDirector project, you can create new tests in WinRunner and save them directly to your project. To save a test, you give it a descriptive name and associate it with the relevant subject in the test plan tree. This helps you to keep track of the tests created for each subject and to quickly view the progress of test planning and creation.

To save a test to a TestDirector project:

1 Choose **File > Save** or click the **Save** button. For a test already saved in the file system, choose **File > Save As**.

The **Save Test to TestDirector Project** dialog box opens and displays the test plan tree.
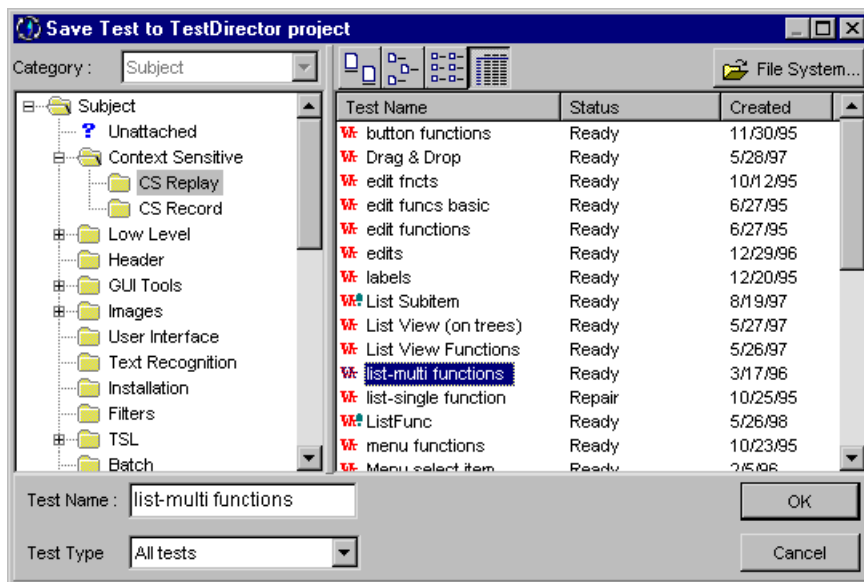
Note that the **Save Test to TestDirector Project** dialog box opens only when WinRunner is connected to a TestDirector project.

To save a test directly in the file system, click the **File System** button, which opens the **Save Test** dialog box. (From the **Save Test** dialog box, you may return to the **Save Test to 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.

---

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 Tests** 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.

**Books
Online**

M Find

**Find
Again**

? Help

**Top of
Chapter**

⇐ Back
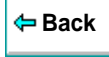
Opening Tests in a Project

If WinRunner is connected to a TestDirector project, 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.

Books Online

Find

Find Again

? Help

To open a test saved to a TestDirector project:

1 Choose **File > Open** or click the **Open** button.

The **Open Test from TestDirector Project** dialog box opens and displays the test plan tree.
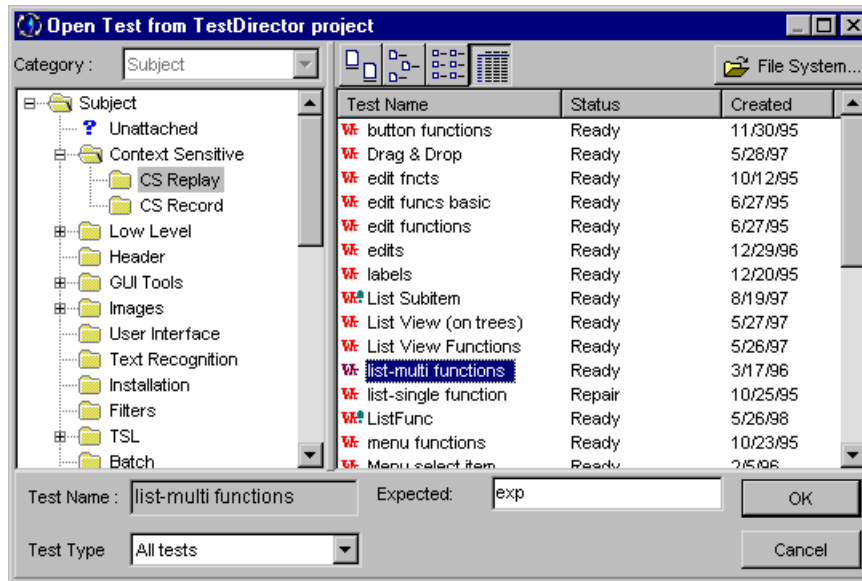
Top of Chapter

← Back

Note that the **Open Test from TestDirector Project** dialog box opens only when WinRunner is connected to a TestDirector project.

Note: If the test you are opening is currently checked into the version control database, then the **latest version** opens and a WinRunner message informs you that you cannot make any changes to this script until you check it out. For more information on version control, see **Managing Test Versions in WinRunner** on page 998.

To open a test directly from the file system, click the **File System** button, which opens the **Open Test** dialog box. (From the **Open Test** dialog box, you 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.

---

2 Click the relevant subject in the test plan tree. To expand the tree and view sublevels, double-click closed folders. To collapse the tree, double-click open folders.

Note that when you select a subject, the tests that belong to the subject appear in the **Test Name** list.

3 Select a test from the **Test Name** list in the right pane. The test appears in the read-only **Test Name** box.

4 If desired, enter an expected results folder for the test in the **Expected** box. (Otherwise, the default folder 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 Tests** in the **Test Type** box. For more information on batch tests, see Chapter 29, **Running Batch Tests**.

Managing Test Versions in WinRunner

When WinRunner is connected to a TestDirector Project with version control support, you can update and revise your automated test scripts while maintaining old versions of each test. This helps you keep track of the changes made to each test script, see what was modified from one version of a script to another, or return to a previous version of the test script.

Note: A TestDirector Project with version control support requires the installation of version control software as well as TestDirector's version control software components. For more information, refer to the *TestDirector Installation Guide.*

You manage test versions by checking tests in and out of the version control database.

Books Online

M Find

Find Again

? Help

Top of Chapter

Back

Adding Tests to the Version Control Database

When you add a test to the version control database for the first time, it becomes the **Working Test** and is also assigned a permanent version number.

The **Working Test** is the test that is located in the test repository and is used by TestDirector for all test runs.

Note: Usually the latest version is the **Working Test**, but any version can be designated as the **Working Test** using the Version Manager application. For more information about the Version Manager, refer to the *TestDirector User's Guide.*

To add a new test to the version control database:

1 Choose **File > Check In.**

Note: The **Check In** and **Check Out** options in the **File** menu are only visible when you are connected to a Test Director Project Database with version control support, and you have a test open. The **Check In** option will be enabled only if the active script has been saved to the project database.

2 Choose **OK** to confirm adding the test to the version control database.

3 Click **OK** to reopen the checked-in test. The test will close and then reopen as a read-only file.

If you have made unsaved changes in the active test, you will be prompted to save the test.

You can review the checked-in test. You can also run the test and view the results. While the test is checked in and is in read-only format, however, you cannot make any changes to the script.

If you attempt to make changes, a WinRunner message reminds you that the script has not been checked out and that you cannot change it.

Books Online

Find

Find Again

Help

Top of Chapter

Back

Checking Tests Out of the Version Control Database

When you open a test which is currently checked in to the version control database, you cannot make any modifications to the script. If you wish to make modifications to this script, you must check out the script.

When you check out a test, the Version Manager copies the **latest version** of the test to your unique checkout directory (automatically created the first time you check out a test), and locks the test in the project database. This prevents other users of the TestDirector project from overwriting any changes you make to the test.
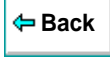
To check out a test:

1 Choose **File > Check Out**.

2 Click **OK**. The read-only test will close and automatically reopen as a writable script.

Note: The **Check Out** option is enabled only if the active script is currently checked in to the project's version control database.

You should check a script out of the version control database only when you want to make modifications to the script or to test the script for workability.

When you run a test while the test is checked out, the results are displayed at the end of the test (if you select the **Display test results at end of run** check box), but they are not saved to the Test Director project database.

When you are ready to run tests on your application, you should always check the script into the version control database so that the test results will be stored in the Test Director project database.

Note: The results which are displayed for a checked-out test only include the tests which have been run since the test was last checked out. When you check in the test, the results of any tests you ran while the test was checked out, will be deleted.

Books Online

M Find

Find Again

? Help

Top of Chapter

← Back

Checking Tests In to the Version Control Database

When you have finished making changes to a test you check it in to the version control database in order to make it the new **latest version** and to assign it as the **Working Test**.

When you check a test back into the version control database, the Version Manager deletes the test copy from your checkout directory and unlocks the test in the database so that the test version will be available to other users of the TestDirector project.

To check in a test:

 1 Choose **File > Check In**.

 2 Click **OK**. The file will close and automatically reopen as a read-only script.

If you run tests after you have checked in the script, the results will be saved to the TestDirector Project database.

Tip: You should close a test in WinRunner before using the Version Manager to change the checked in/checked out status of the test. If you make changes to the test's status via Version Manager while the test is open in WinRunner, WinRunner will not reflect those changes. For more information about Version Manager, refer to the *TestDirector User's Guide.*

Saving GUI Map Files to a Project

When WinRunner is connected to a TestDirector project, choose File > Save in the GUI Map Editor to save your GUI map file to the open database. All the GUI map files used in all the tests saved to the TestDirector project are stored together. This facilitates keeping track of the GUI map files associated with tests in your project.

To save a GUI map file to a TestDirector project:

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

2 From a temporary GUI map file, choose **File > Save**. From an existing GUI map file, choose **File > Save As**.

The **Save GUI File to 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.
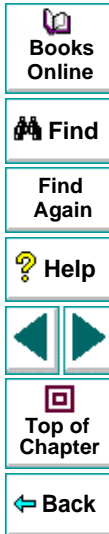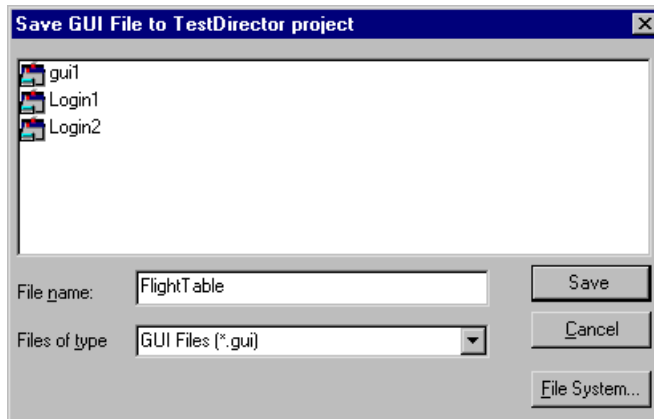
Note that the **Save GUI File to TestDirector Project** dialog box opens only when WinRunner is connected to a TestDirector project.

To save a GUI map file directly to the file system, click the **File System** button, which opens the **Save GUI File** dialog box. (From the **Save GUI File** dialog box, you may return to the **Save GUI File to 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.

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 Project

When WinRunner is connected to a TestDirector project, 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:

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.

Open GUI File from TestDirector project	☒

gui1
Login1
Login2

File name:	Login1	Open
Files of type	GUI Files (*.gui)	Cancel

⊙ Load into the GUI Map   ○ Open For Editing Only   File System...
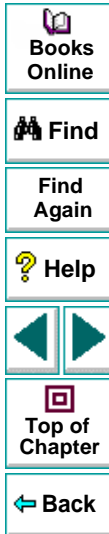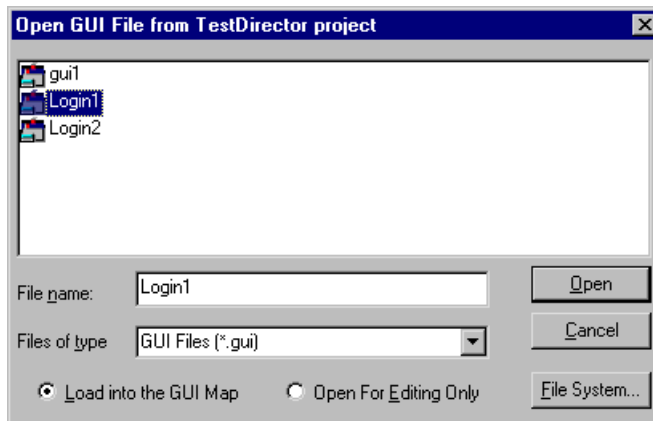
Note that the **Open GUI File from TestDirector Project** dialog box opens only when WinRunner is connected to a TestDirector project.

To open a GUI map file directly from the file system, click the **File System** button, which opens the **Open GUI File** dialog box. (From the **Open GUI File** dialog box, you may return to the **Open GUI File from 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** 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 and you want to run tests in the project from WinRunner, specify the name of the current test set before you begin. When the test run is completed, the tests are stored in 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
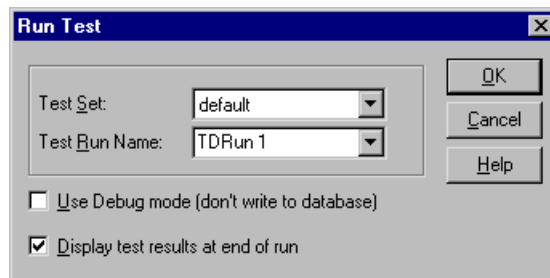
2 In the **Test Set** box, select a test set from the list. The list contains test sets created in TestDirector.

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.

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 on Remote Hosts

If you are using the client/server edition of TestDirector, you can run WinRunner tests on multiple remote hosts. To enable TestDirector to use a computer as a remote host, you must activate the Allow TestDirector to Run Tests Remotely option. Note that when you run a test on a remote host, you should run the test in silent mode, which suppresses WinRunner messages during a test run. For more information on silent mode, see Chapter 37, **Setting Testing Options from a Test Script**.

To enable TestDirector on a remote machine to run WinRunner tests:

1 Choose **Settings > General Options** to open the **General 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 General Options dialog box, see Chapter 36, **Setting Global Testing Options**.

Viewing Test Results from a Project

If you run tests in a test set, you can view the test results from a TestDirector project. If you run a test set in Verify mode, the Test Results window opens automatically at the end of the test run. At other times, choose Tools > Test Results to open the Test Results window. By default, the Test Results window displays the test results of the last test run of the active test. To view the test results for another test or for an earlier test run of the active test, choose File > Open in the Test Results window.

To view test results from a TestDirector project:

1 Choose **Tools > Test Results**.

The **Test Results** window opens, displaying the test results of the last test run of
the active test.

2 In the **Test Results** window, choose **File > Open**.

The **Open Test Results from TestDirector Project** dialog box opens and
displays the test plan tree.

Open Test Results from TestDirector project				
Category:	Subject			File System...
		Run Name	Cycle	Status	Run Date

win_functions	✗ Run(6/18 15:...	WR_5_0	Not Completed	6/18/98 3:40:39 PM
button functions	✓ Tuborg95	WR_5_BETA	Passed	2/16/98 8:40:31 PM
Spin32 functions				
Menu select item				
Tab functions				
Toolbar function:				
Replay by Descri				
list-multi function				
Popup menu				
_Return Values				
edits				
Non Editable edit				
Win95 contrl				
Drag & Drop				

Test Name: list-multi functions   Run Name: [   ]   OK

Test Type: All tests   Cancel

Books Online

Find

Find Again

Help

Top of Chapter

Back
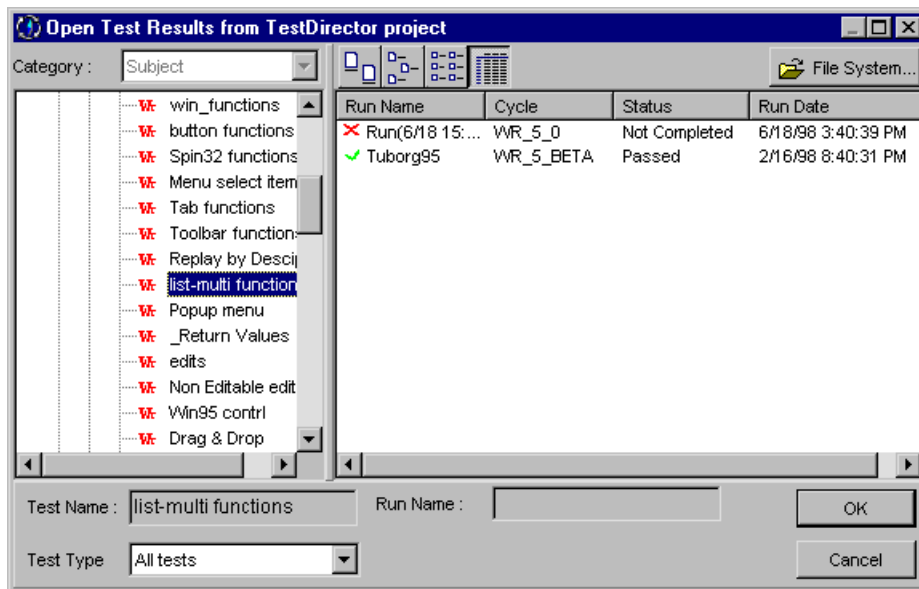
Note that the **Open Test Results from TestDirector Project** dialog box opens only when WinRunner is connected to a TestDirector project.

To open test results directly from the file system, click the **File System** button, which opens the **Open Test Results** dialog box. (From the **Open Test Results** dialog box, you may return to the **Open Test Results from 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. In the right pane:

- The **Run Name** column displays whether your test run passed or failed and contains the names of the test runs.

- The **Test Set** column contains the names of the test sets.

- Entries in the **Status** column indicate whether the test passed or failed.

- The **Run Date** column displays the date and time when the test set was run.

6 Click **OK** to view the results of the selected test.

For information about the options in the Test Results window, see Chapter 28, **Analyzing Test Results**.

Using TSL Functions with TestDirector

Several TSL functions facilitate your work with a TestDirector project by returning the values of fields in a TestDirector project. 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 in a TSL statement instead of using the full file system path.

TestDirector Project Functions

Several TSL functions enable you to retrieve information from a TestDirector project.

tddb_get_step_value	Returns the value of a field in the "dessteps" table in a TestDirector project.
tddb_get_test_value	Returns the value of a field in the "test" table in a TestDirector project.
tddb_get_testset_value	Returns the value of a field in the "testcycl" table in a TestDirector project.

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*.

Books Online

Find

Find Again

Help

Top of Chapter

Back

Call Statements 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 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, 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 General Options dialog box or by using a **setvar** statement in your test script, you can call the test from your test script with the following statement:

call ("My_test");

Note that the [TD] prefix is optional when specifying a test or a compiled module in a TestDirector project.

Note: When you run a WinRunner test from a TestDirector project, 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*.

Books Online

Find

Find Again

Help

Top of Chapter

Back

GUI Map Editor Functions

When WinRunner is connected to TestDirector, you can specify the names of GUI map files saved in a TestDirector project when you use GUI Map Editor functions in a test script.

When WinRunner is connected to a TestDirector project, 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.

For example, if the My_gui.gui GUI map file is stored in a TestDirector project, in My_project_database\GUI, you can load it with the statement:

GUI_load ("My_gui.gui");

For information about working with GUI Map Editor functions, refer to the *TSL Online Reference*.

Books Online

Find

Find Again

? Help

Top of Chapter

← Back

Specifying Search Paths for Tests Called from TestDirector

You can configure WinRunner to use search paths based on the path in a TestDirector project.

In the following example, a **setvar** statement specifies a search path in a TestDirector project:

setvar (searchpath, [TD]\My_project_database\Subject\Sub1);

For information on how to specify the search path using the General Options dialog box, see Chapter 36, **Setting Global Testing Options**. For information on how to specify the search path by using a **setvar** statement, see Chapter 37, **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.

Note that you can use the corresponding *td_connection* testing option to activate or deactivate WinRunner's connection to TestDirector, as described in Chapter 37, **Setting Testing Options from a Test Script**.

Note that you can connect to TestDirector from the **Connection to TestDirector** dialog box, which you open by choosing Tools > TestDirector Connection. For more information about connecting to TestDirector, see **Connecting to and Disconnecting from a Project** on page 986.

-td_cycle_name *cycle_name*

Specifies the name of the current test cycle. This option is applicable only when WinRunner is connected to TestDirector.

Note that you can use the corresponding *td_cycle_name* testing option to specify the name of the current test cycle, as described in Chapter 37, **Setting Testing Options from a Test Script**.

-td_database_name *database_pathname*

Specifies the active TestDirector project. WinRunner can open, execute, and save tests in this project. This option is applicable only when WinRunner is connected to TestDirector.

Note that you can use the corresponding *td_database_name* testing option to specify the active TestDirector database, as described in Chapter 37, **Setting Testing Options from a Test Script**.

Note that when WinRunner is connected to TestDirector, you can specify the active TestDirector project from the TestDirector Connection dialog box, which you open by choosing **Tools > TestDirector Connection**. For more information about connecting to TestDirector, see **Connecting to and Disconnecting from a Project** on page 986.

-td_logname_dir *event log file path*

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 project in a TestDirector server.

Note that you can specify the password for connecting to TestDirector from the **Connection to TestDirector** dialog box, which you open by choosing **Tools > TestDirector Connection**. For more information about connecting to TestDirector, see **Connecting to and Disconnecting from a Project** on page 986.

-td_server_name

Specifies the name of the TestDirector server to which WinRunner connects.

Note that you can use the corresponding *td_server_name* testing option to specify the name of the TestDirector server to which WinRunner connects, as described in Chapter 37, **Setting Testing Options from a Test Script**.

Note that you can specify the name of the TestDirector server to which WinRunner connects from the **Connection to TestDirector** dialog box, which you open by choosing **Tools > TestDirector Connection**. For more information about connecting to TestDirector, see **Connecting to and Disconnecting from a Project** on page 986.

Books Online

Find

Find Again

Help

Top of Chapter

Back

-td_user_name *user_name*

Specifies the name of the user who is currently executing a test cycle. (Formerly *user.*)

Note that you can use the corresponding *td_user_name* testing option to specify the user, as described in Chapter 37, **Setting Testing Options from a Test Script**.

Note that you can specify the user name when you connect to TestDirector from the **Connection to TestDirector** dialog box, which you open by choosing **Tools > TestDirector Connection**. For more information about connecting to TestDirector, see **Connecting to and Disconnecting from a Project** on page 986.

For more information on using command line options, see Chapter 30, **Running Tests from the Command Line**.

Books Online

Find

Find Again

Help

Top of Chapter

Back

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:

- **Emulating Multiple Users**

- **Virtual User (Vuser) Technology**

- **Developing and Running Scenarios**

- **Creating GUI Vuser Scripts**

- **Measuring Server Performance**

- **Synchronizing Virtual User Transactions**

- **Creating a Rendezvous Point**

- **A Sample Vuser Script**

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.
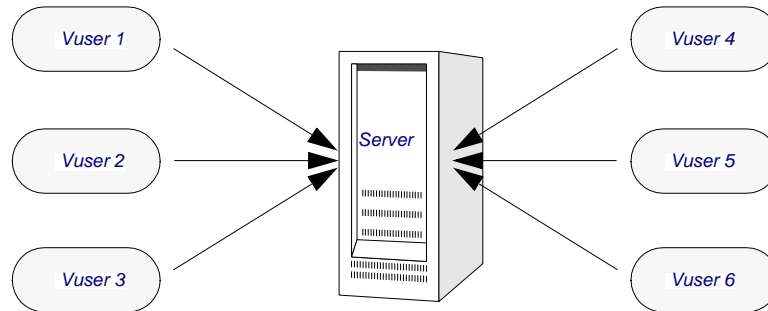
Load testing your server with LoadRunner

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.
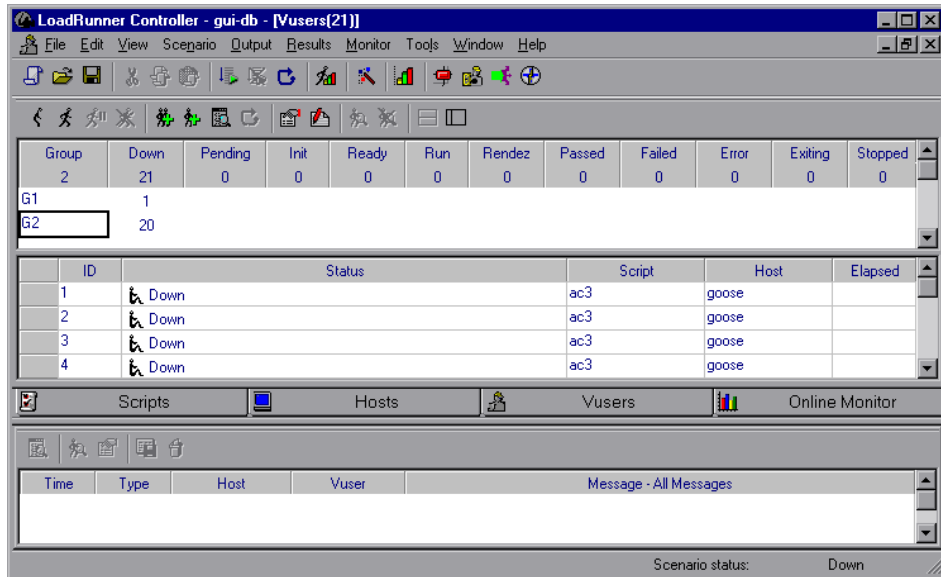
The following procedure outlines how to use the LoadRunner Controller to create, run, and analyze a scenario. For more information, refer to the **LoadRunner Controller User's Guide**.

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 Start WinRunner.

 2 Start the client application.

 3 Record operations on the client application.

 4 Edit the Vuser script using WinRunner, and program additional TSL statements. Add control-flow structures as needed.

 5 Define actions within the script as transactions to measure server performance.

 6 Add synchronization points to the script.

 7 Add *rendezvous* points to the script to coordinate the actions of multiple Vusers.

 8 Save the script and exit WinRunner.

Measuring Server Performance

Transactions measure how your server performs under the load of many users. A transaction may be a simple task, such as entering text into a text field, or it may be an entire test that includes multiple tasks. LoadRunner measures the performance of a transaction under different loads. You can measure the time it takes a single user or a hundred users to perform the same transaction.

The first stage of creating a transaction is to declare its name at the start of the Vuser script. When you assign the Vuser script to a Vuser, the Controller scans the Vuser script for transaction declaration statements. If the script contains a transaction declaration, LoadRunner reads the name of the transaction and displays it in the Transactions window.

Books Online

Find

Find Again

? Help

Top of Chapter

⇐ Back

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

declare_transaction ("*transaction_name*" **);**

The *transaction_name* must be a string constant, not a variable or an expression. This string can contain up to 128 characters. No spaces are permitted.

Next, mark the point where LoadRunner will start to measure the transaction. Insert a **start_transaction** statement into the Vuser script immediately before the action you want to measure. The syntax of this function is:

start_transaction ("*transaction_name*" **);**

The *transaction_name* is the name you defined in the **declare_transaction** statement.

Insert an **end_transaction** statement into the Vuser script to indicate the end of the transaction. If the entire test is a single transaction, then insert this statement in the last line of the script. The syntax of this function is:

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

The *transaction_name* is the name you defined in the **declare_transaction** statement. The *status* tells LoadRunner to end the transaction only if the transaction passed (PASS) or failed (FAIL).

Synchronizing 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 17, **Synchronizing the Test Run**.

Books Online

Find

Find Again

Help

Top of Chapter

Back

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");

Books
Online

Find

Find
Again

Help

Top of
Chapter

Back

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);

Books Online

Find

Find Again

Help

Top of Chapter

Back

You can report defects detected in your application using the Remote Defect Reporter.

This chapter describes:

- **Using the Web Defect Manager**
- **Setting Up the Remote Defect Reporter**
- **The Remote Defect Reporter Window**
- **Reporting New Defects from the Remote Defect Reporter**

About Reporting Defects

You can use the Web Defect Manager or the Remote Defect Reporter to report new defects detected in your application. If the TestDirector Web Defect Manager is installed on your machine, and you are already using it, then it opens directly from the WinRunner Test Results Window. Otherwise, the Remote Defect Reporter setup message is displayed. If you set up the Remote Defect Reporter on your machine, then it 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.

You can use TestDirector to determine which fields appear in the Remote Defect Reporter by customizing the database. For more information about customizing the database, refer to the *TestDirector Administrator's Guide*.

Books Online

M Find

Find Again

? Help

Top of Chapter

Back

Using the Web Defect Manager

The Web Defect Manager is Mercury Interactive's system for reporting and tracking software defects and errors over the World Wide Web. The Web Defect Manager is a scalable, defect tracking system that helps you monitor defects closely from initial detection until resolution.

The Web Defect Manager is tightly integrated with TestDirector, Mercury Interactive's test management tool. Multiple users can share defect-tracking information stored in a central repository (TestDirector project). Several projects can be stored on a database server. This ensures that all software development, Quality Assurance, and Information Systems personnel can share defect-tracking information. For more information about TestDirector projects, refer to the *TestDirector User's Guide*.

When you detect a defect in your application, you report it to a TestDirector project. A TestDirector project stores defect information in a central defect repository that can be accessed by members of the development, quality assurance, and support teams.

Books Online

Find

Find Again

Help

Top of Chapter

Back

For example, suppose you are testing a travel agency application. You discover that errors occur when you try to order an airline ticket. You open and report the defect. This includes a summary and detailed description of the defect, where it was discovered, and if you are able to reproduce it. The report can also include screen captures, text documents, and other files relevant to understanding and repairing the problem. For information on using the TestDirector Web Defect Manager, refer to the *Web Defect Manager User's Guide*.

Before you can launch the Web Defect Manager, you must ensure that a Web browser is installed on your computer. You must also ensure that a Web Defect Manager Server is installed on your Web server. For more information about the Web Defect Manager Server, refer to the *TestDirector Installation Guide*.

Note: If you want to use with Netscape Navigator, you must run the Netscape Plug-in installation. For more information, refer to the *TestDirector Installation Guide*.

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 install the Remote Defect Reporter from the TestDirector program group.

To set up the Remote Defect Reporter from WinRunner:

 1 Make sure that the Test Results window is open. If necessary, choose **Tools > Test Results** or click the **Test Results** button to open it.

 2 Choose **Tools > Report Bug** or click the **Report Bug** button.

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.

Books Online

Find

Find Again

Help

Top of Chapter

Back
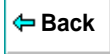
5 Choose the configuration folder, where the files for determining which fields appear in the Remote Defect Reporter are stored. The default location for this folder is the TDPriv folder. To change the location of the configuration folder, click the **Browse** button, select the desired location, and 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 Defect > Run Setup in the Remote Defect Reporter window.
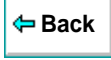
The Remote Defect Reporter Window

Once the Remote Defect Reporter has been set up, you can use it to report bugs.

To open the Remote Defect Reporter window:

1 Choose **Tools > Test Results** or click the **Test Results** button to open the WinRunner **Test Results** window.

2 Choose **Tools > Report Bug** or click the **Report Bug** button.

The **Remote Defect Reporter** window opens.

Reporting New Defects from the Remote Defect Reporter

You can report defects detected in your application directly from the WinRunner
Test Results window.

To report new defects:

1 Make sure that the Test Results window is open. If necessary, choose
Tools > Test Results or click the **Test Results** button to open it.

2 Choose **Tools > Report Bug** or click the **Report Bug** button.

3 The **Remote Defect Reporter** window opens.

4 Under **Detected By**, click the name of the person who detected the defect. Note
that the current date appears automatically.

5 Type in a brief summary of the defect.

6 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.

7 Enter test reference information about the defect in the **Attachments** section.

8 Choose either **Defect > Save to File** or **Defect > Deliver via E-mail**, depending on the option you chose during setup. Alternatively, click the **Save to File** button or the **Deliver via E-mail** button. 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 your machine is not properly configured 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.

Index

Books Online

Find

Find Again

Help

Top of Chapter

Back

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

Books Online

Find

Find Again

Help

Top of Chapter

Back

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

Books Online

M Find

Find Again

? Help

Top of Chapter

← Back

Books Online

M Find

Find Again

? Help

◀ ▶

Top of Chapter

⇐ Back

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

Books Online

Find

Find Again

Help

Top of Chapter

Back

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

Books Online

Find

Find Again

Help

Top of Chapter

Back

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 **page 1056**

Books Online

Find

Find Again

Help

Top of Chapter

Back

Books
Online

A Find

Find
Again

? Help

Top of
Chapter

← Back

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

Books Online

Find

Find Again

Help

Top of Chapter

Back

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

Books Online

Find

Find Again

Help

Top of Chapter

Back

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

Books Online

Find

Find Again

Help

Top of Chapter

Back

Books Online

Find

Find Again

Help

Top of Chapter

Back

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

Books Online

Find

Find Again

Help

Top of Chapter

Back

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

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

**Books
Online**

M Find

**Find
Again**

? Help

**Top of
Chapter**

⇐ Back

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

Books Online

Find

Find Again

Help

Top of Chapter

Back

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

Books Online

Find

Find Again

Help

Top of Chapter

Back

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

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

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

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

Books Online

M Find

Find Again

? Help

◀ ▶

Top of Chapter

← Back

Books Online

Find

Find Again

Help

Top of Chapter

Back

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

Books Online

Find

Find Again

Help

Top of Chapter

Back

Books Online

Find

Find Again

Help

Top of Chapter

Back

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

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

Books Online

Find

Find Again

Help

Top of Chapter

Back

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

Books Online

Find

Find Again

Help

Top of Chapter

Back

Books Online
Find
Find Again
Help
Top of Chapter
Back

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

Books Online

Find

Find Again

Help

Top of Chapter

Back

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

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

Books Online

M Find

Find Again

? Help

Top of Chapter

← Back

Books Online

Find

Find Again

Help

Top of Chapter

Back

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

Books Online

Find

Find Again

Help

Top of Chapter

Back

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

Books Online

Find

Find Again

Help

Top of Chapter

Back

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

Index

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

Books Online

Find

Find Again

Help

Top of Chapter

Back

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

XRunner
   bitmap checkpoints **406**
   GUI checkpoints **191**
   GUI maps **138**
   tests recorded in Analog mode **165**

Y

y property **129**, **141**

Books Online

Find

Find Again

Help

Top of Chapter

Back

WinRunner User's Guide, Version 6.0

© Copyright 1994 - 1999 by Mercury Interactive Corporation

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

WRUG6.0/01