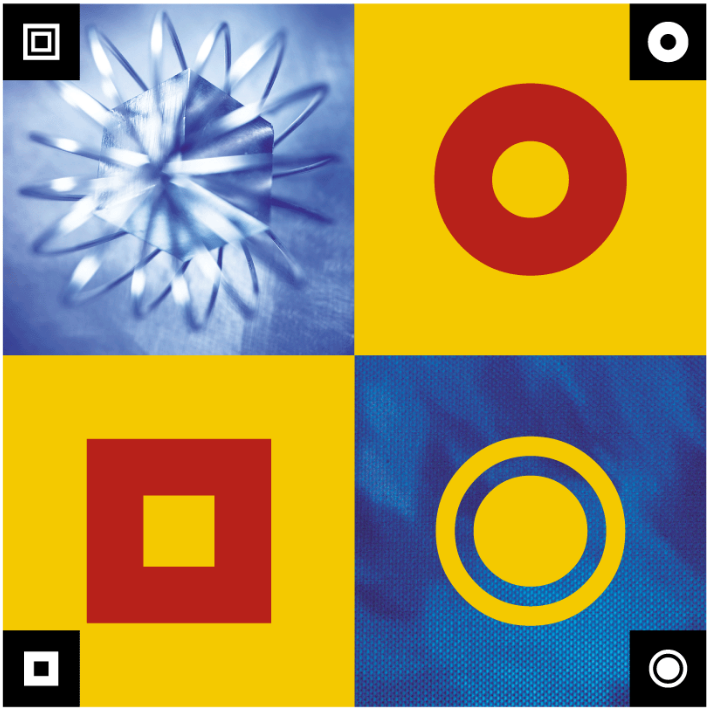
MERCURY INTERACTIVE

WinRunner



USER'S GUIDE

WinRunner

User's Guide

Version 7.6

MERCURY INTERACTIVE

WinRunner User's Guide, Version 7.6

Mercury Interactive Corporation
1325 Borregas Avenue
Sunnyvale, CA 94089 USA
Tel: (408) 822-5200
Toll Free: (800) TEST-911, (866) TOPAZ-4U
Fax: (408) 822-5300

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

WRUG7.6/01

Contents Summary

PART I: STARTING THE TESTING PROCESS

PART II: UNDERSTANDING THE GUI MAP

PART III: CREATING TESTS

PART IV: PROGRAMMING WITH TSL

PART V: RUNNING TESTS

PART VI: DEBUGGING TESTS

PART VII: CONFIGURING WINRUNNER

PART VIII: WORKING WITH OTHER MERCURY INTERACTIVE PRODUCTS

Table of Contents

PART I: STARTING THE TESTING PROCESS

PART II: UNDERSTANDING THE GUI MAP

PART III: CREATING TESTS

PART IV: PROGRAMMING WITH TSL

PART V: RUNNING TESTS

PART VII: CONFIGURING WINRUNNER

PART VIII: WORKING WITH OTHER MERCURY INTERACTIVE PRODUCTS

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, insert checkpoints, assign parameters, use regular expressions, and handle unexpected events that occur during a test run.

Part IV Programming with TSL

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

Part V Running Tests

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

Part VI Debugging Tests

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

Part VII Configuring WinRunner

Describes how to customize WinRunner’s user interface, test script editor and the Function Generator. You can also change WinRunner’s default settings, both globally and per test, and initialize special configurations to adapt WinRunner to your testing environment.

Part VIII Working with Other Mercury Interactive Products

Describes how to integrate with QuickTest Professional, report defects detected in your application, and how WinRunner interacts with TestDirector and LoadRunner.

WinRunner Documentation Set

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

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

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

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

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

Online Resources

WinRunner includes the following online resources:

Read Me 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. It is recommended that you use version 5.0 or later. You can download Adobe Acrobat Reader from www.adobe.com. 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 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 Reference*.

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

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

Support Information presents Mercury Interactive's home page, its Customer Support Web site, and other Web addresses to help you contact Mercury Interactive's offices around the world.

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

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.

Arial The Arial 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, an ellipsis indicates that more items of the same format may be included. In a program example, an ellipsis is 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.

Part I

Starting the Testing Process

1

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.

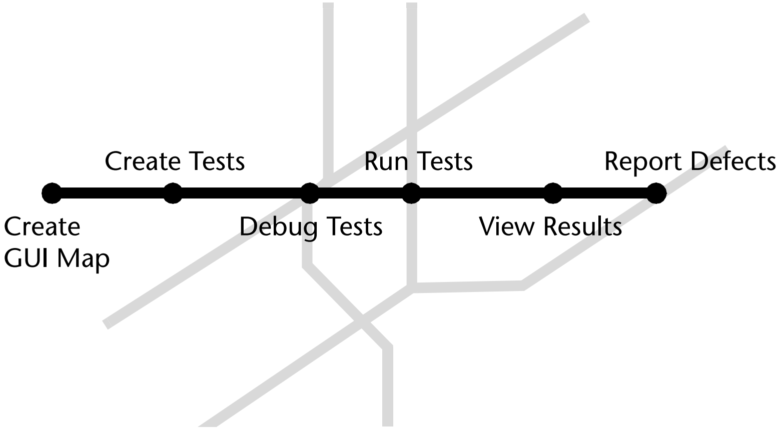
To run a test, you simply play back the test script. WinRunner emulates a user by moving the mouse pointer over your application, selecting objects, and entering keyboard input. WinRunner reads the object descriptions in the GUI map and then searches in the application being tested for objects matching 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 the GUI Map

The first stage is to create the GUI map so WinRunner can recognize the GUI objects in the 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.

Note that when you work in *GUI Map per Test* mode, you can skip this step. For additional information, see Chapter 3, "Understanding How WinRunner Identifies GUI Objects."

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've finished debugging the test.

When WinRunner runs a test, it checks each script line for basic syntax errors, like incorrect syntax or missing elements in **If**, **While**, **Switch**, and **For** statements. You can use the **Syntax Check** options (**Tools** >**Syntax Check**) to check for these types of syntax errors before running your 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.

You can view your results in the standard WinRunner report view or in the Unified report view. The WinRunner report view displays the test results in a Windows-style viewer. The Unified report view displays the results in an HTML-style viewer (identical to the style used for QuickTest Professional test 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.

You can also insert **tddb_add_defect** statements to your test script that instruct WinRunner to add a defect to a TestDirector project based on conditions you define in your test script.

Sample Applications

Many examples in this book use the sample Flight Reservation application provided with WinRunner.

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 4A or Flight 4B.

Multiple Versions of the Sample Application

The sample Flight Reservation application comes in two versions: Flight 4A and Flight 4B. Flight 4A is a fully working application, while Flight 4B 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 4A or Flight 4B.

When WinRunner is installed with Visual Basic support, Visual Basic versions of Flight 4A and Flight 4B applications are installed in addition to the regular sample applications.

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.

Sample Web Application

WinRunner also includes a sample flight reservation application for the Web. The URL for this Web site is *http://newtours.mercuryinteractive.com*. You can also start this application by choosing **Start** > **Programs** > **WinRunner** > **Sample Applications** > **Mercury Tours site**.

Integrating with other Mercury Interactive Products

WinRunner works with other Mercury Interactive products 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.

QuickTest Professional

QuickTest Professional is an easy to use, yet comprehensive, icon-based functional testing tool designed to perform functional and regression testing of dynamic Windows-based, Visual Basic, ActiveX, Web, and multimedia applications. You can also expand QuickTest's functionality to test your applications created using leading-edge development environments such as Java, .NET, SAP, Siebel, PeopleSoft, and Oracle.

You can design tests in QuickTest Professional and then leverage your investments in existing WinRunner script libraries by calling WinRunner tests and functions from your QuickTest test. You can also call QuickTest tests from WinRunner.

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.

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.

2

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

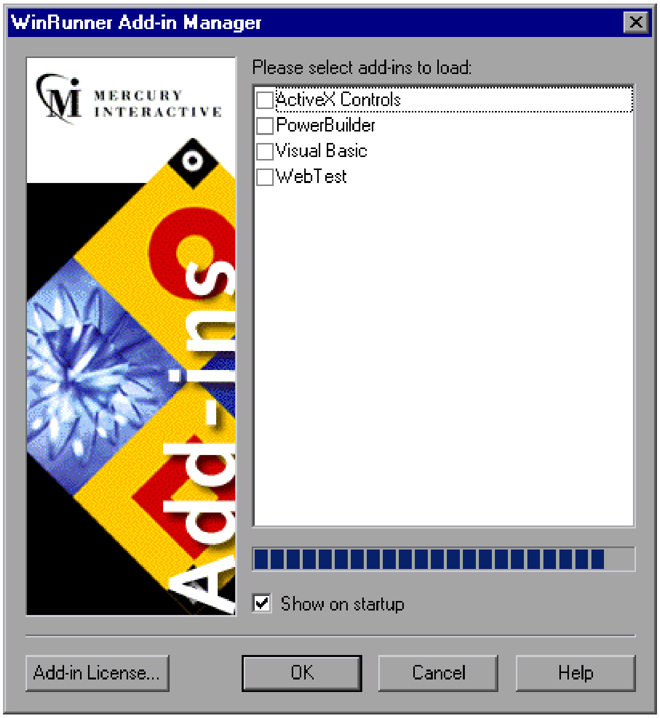
Starting WinRunner

To start WinRunner:

Choose **Programs** > **WinRunner** > **WinRunner** on the **Start** menu.

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 WinRunner Add-in Manager dialog box opens.



WinRunner Add-in Manager

Please select add-ins to load:

- □ ActiveX Controls
- □ PowerBuilder
- □ Visual Basic
- □ WebTest

☑ Show on startup

Note: The first time you start WinRunner, "What's New in WinRunner" help also opens.

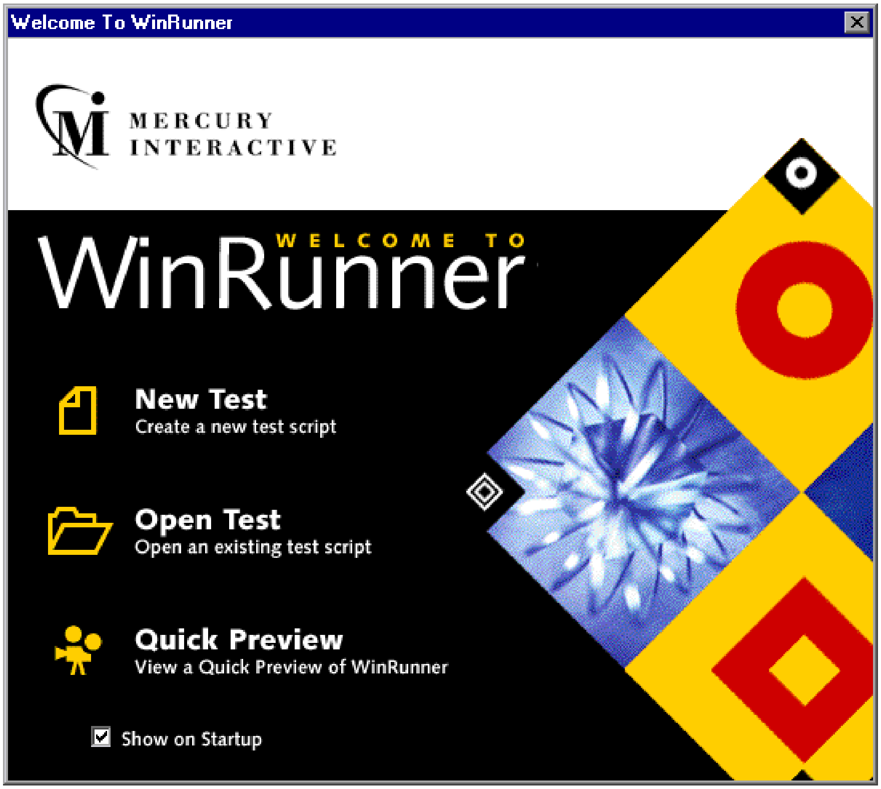
The WinRunner Add-in Manager dialog box contains a list of the add-ins available on your computer. The WinRunner installation includes the ActiveX Controls, PowerBuilder, Visual Basic, and WebTest add-ins.

You can also extend WinRunner's functionality to support a large number of development environments by purchasing external WinRunner add-ins. If you install external WinRunner add-ins they are displayed in the Add-in Manager together with the core add-ins. When you install external add-ins, you must also install a special WinRunner add-in license.

The first time you open WinRunner after installing an external add-in, the Add-in Manager displays the add-in, but the check box is disabled and the add-in name is grayed. Click the **Add-in License** button to install the Add-in license.

Select the add-ins you want to load for the current session of WinRunner. If you do not make a change in the Add-in Manager dialog box within a certain amount of time, the window closes and the selected add-ins are automatically loaded. The progress bar displays how much time is left before the window closes.

The Welcome to WinRunner window opens. From the Welcome to WinRunner window you can click **New Test** to create a new test, click **Open Test** to open an existing test, or click **Quick Preview** to view an overview of WinRunner in your default browser.



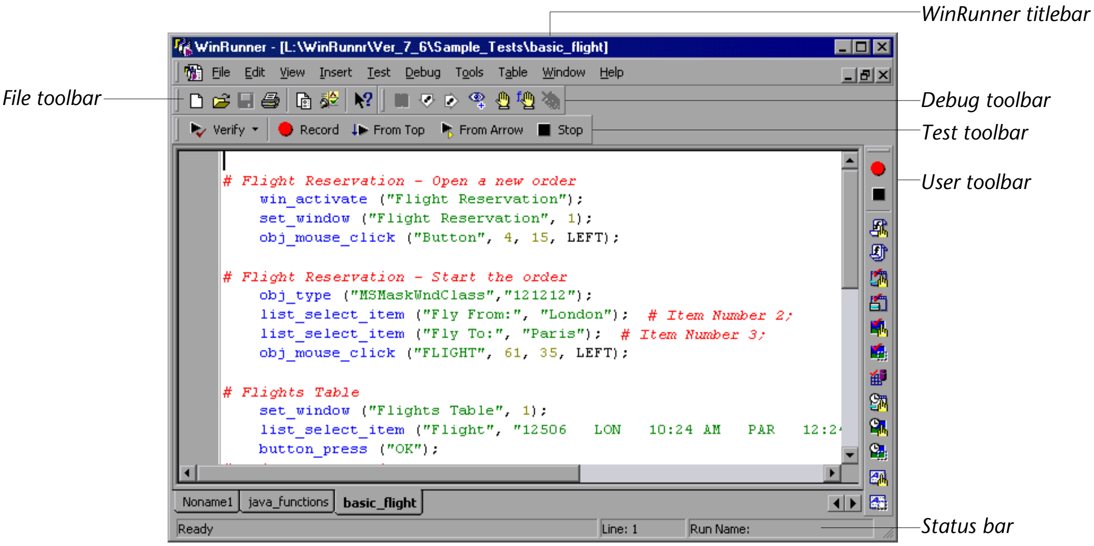
Tip: If you do not want the Welcome to WinRunner window to appear the next time you start WinRunner, clear the **Show on Startup** check box. To show the Welcome to WinRunner window upon startup from within WinRunner, choose **Tools** > **General Options**, select the **General** > **Startup** category, and select the **Display Welcome screen on startup** check box.

The Main WinRunner Window

The main WinRunner window contains the following key elements:

- WinRunner title bar—displays the name and path of the currently open test.
- File toolbar—provides easy access to frequently performed tasks, such as opening and saving tests, and viewing test results.
- Debug toolbar—provides easy access to buttons used while debugging tests.
- Test toolbar—provides easy access to buttons used while running and maintaining tests.
- User toolbar—displays the tools you frequently use to create test scripts. By default, the User toolbar is hidden. To display the Debug toolbar, choose **View** > **User Toolbar**.

- Status bar—displays information on the current command, the line number of the insertion point, and the name of the current results folder



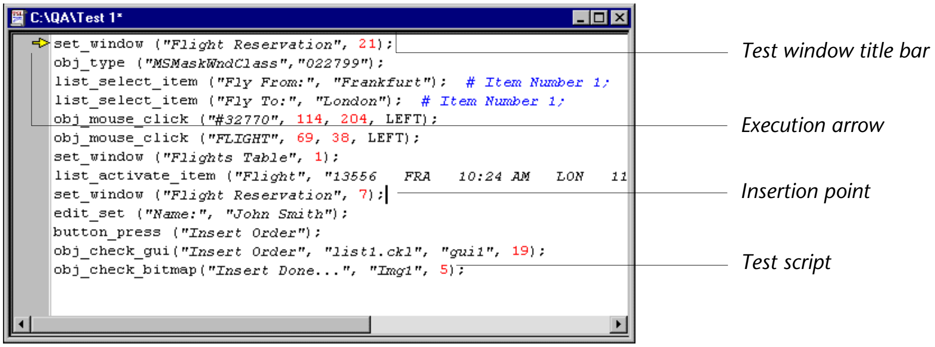
```
# Flight Reservation - Open a new order
    win_activate ("Flight Reservation");
    set_window ("Flight Reservation", 1);
    obj_mouse_click ("Button", 4, 15, LEFT);

# Flight Reservation - Start the order
    obj_type ("MSMaskWndClass","121212");
    list_select_item ("Fly From:", "London");  # Item Number 2;
    list_select_item ("Fly To:", "Paris");  # Item Number 3;
    obj_mouse_click ("FLIGHT", 61, 35, LEFT);

# Flights Table
    set_window ("Flights Table", 1);
    list_select_item ("Flight", "12506   LON   10:24 AM   PAR   12:2
    button_press ("OK");
```

Tip: Each test is displayed in a separate tab. If there are more test tabs than can fit across the bottom of the main WinRunner window, you can click the left or right arrow buttons to scroll the test tabs to the left or the right.

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 during a test run, or the line that will next run if you select the Run from arrow option
- *Insertion point,* which indicates where you can insert or edit text



```
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);
```

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 four built-in toolbars: the *File toolbar*, the *Test toolbar, the Debug toolbar,* and the *User toolbar.* You can customize the *User toolbar* with the commands you use most frequently.

Creating a Floating Toolbar

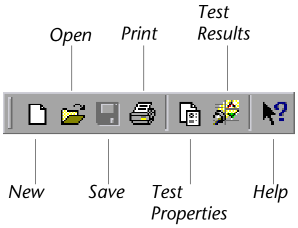
You can change any toolbar to a floating toolbar. In addition, while the User toolbar is a floating toolbar, you can minimize WinRunner and still maintain access to the commands on the User toolbar, so you can work freely with the application being tested.

Double-click a toolbar handle to change it to a floating toolbar; double-click a floating toolbar title bar to snap it back into the toolbar area. You can also drag a toolbar handle or title bar to toggle it from a docked toolbar to a floating toolbar and vice versa.

The File Toolbar

The File toolbar contains buttons for the commands used for frequently performed tasks, such as opening and saving tests, viewing test results, and accessing help. The default location of the File toolbar is docked below the WinRunner menu bar.

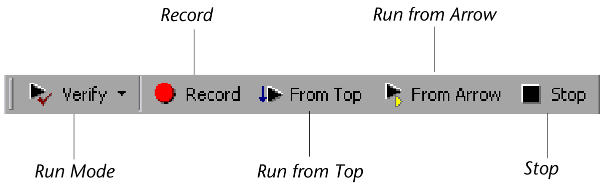
For more information about the File toolbar, see Chapter 11, “Designing Tests.” The following buttons appear on the File toolbar:



The Test Toolbar

The Test toolbar contains buttons for the commands used in running a test. The default location of the Test toolbar is docked below the WinRunner File toolbar.

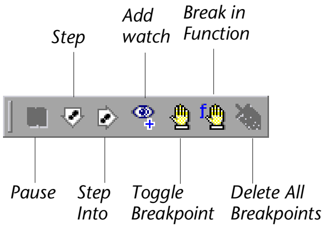
For more information about the Test toolbar, see Chapter 33, "Understanding Test Runs." The following buttons appear on the Test toolbar:



The Debug Toolbar

The Debug toolbar contains buttons for commands used while debugging tests. The default location of the Debug toolbar is docked below the WinRunner menu bar, to the right of the File toolbar.

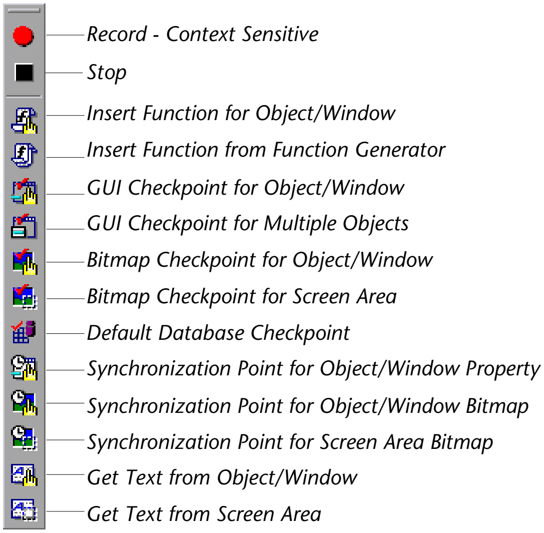
For more information about the Debug toolbar, see Chapter 33, "Understanding Test Runs." The following buttons appear on the Debug toolbar:



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 **View** > **User Toolbar**. 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 839. The following buttons appear by default on the User toolbar:


Record - Context Sensitive
Stop
Insert Function for Object/Window
Insert Function from Function Generator
GUI Checkpoint for Object/Window
GUI Checkpoint for Multiple Objects
Bitmap Checkpoint for Object/Window
Bitmap Checkpoint for Screen Area
Default Database Checkpoint
Synchronization Point for Object/Window Property
Synchronization Point for Object/Window Bitmap
Synchronization Point for Screen Area Bitmap
Get Text from Object/Window
Get Text from Screen Area

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.

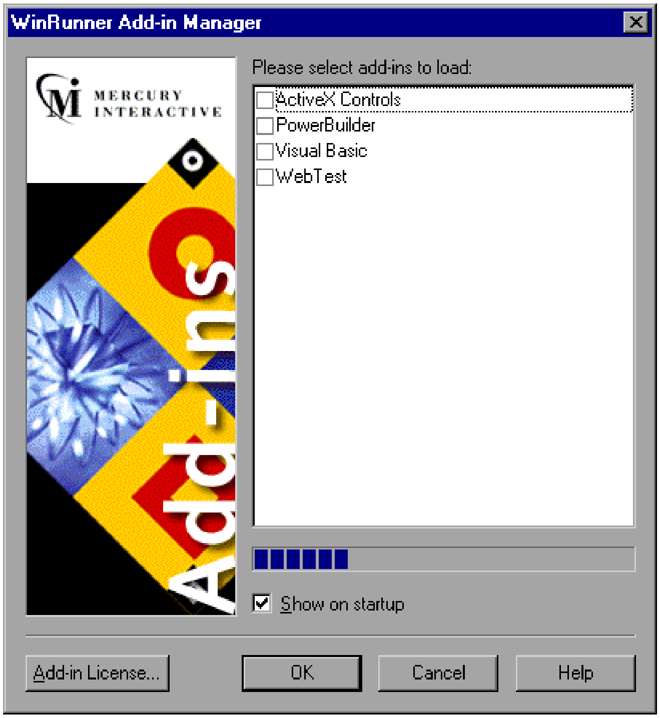
For a list of default WinRunner softkey configurations and information about redefining WinRunner softkeys, see "Configuring WinRunner Softkeys" on page 850.

Loading WinRunner Add-Ins

If you installed add-ins such as WebTest (support for Web sites), 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 and the selected add-ins are automatically loaded.

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 Add-In Manager on startup** option in the **General** > **Startup** category of the General Options dialog box. For information on working with the General Options dialog box, see Chapter 41, “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 36, “Running Tests from the Command Line.”

Part II

Understanding the GUI Map

3

Understanding How WinRunner Identifies GUI Objects

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

This chapter describes:

- About Identifying GUI Objects
- How a Test Identifies GUI Objects
- Physical Descriptions
- Logical Names
- The GUI Map
- Setting the Window Context

About Identifying GUI Objects

When you work in Context Sensitive mode, you can 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. Note that in Context Sensitive mode, WinRunner does not need to know the physical location of a GUI object to identify it.

You can use the GUI Spy to view the properties of any GUI object on your desktop, to see how WinRunner identifies it. For additional information on viewing the properties of GUI objects and teaching them to WinRunner, see Chapter 4, "Understanding Basic GUI Map Concepts."

WinRunner stores the information it learns in a *GUI map*. When WinRunner runs a test, it uses the GUI map to locate objects: 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.

The GUI map is actually the sum of one or more *GUI map files*. There are two modes for organizing GUI map files:

- You can create a GUI map file for your entire application, or for each window in your application. Multiple tests can reference a common GUI map file. This is the default mode in WinRunner. For experienced WinRunner users, this is the most efficient way to work. For more information about working in the *Global GUI Map File* mode, see Chapter 5, "Working in the Global GUI Map File Mode."
- WinRunner can automatically create a GUI map file for each test you create. You do not need to worry about creating, saving, and loading GUI map files. If you are new to WinRunner, this is the simplest way to work. For more information about working in the *GUI Map File per Test* mode, see Chapter 6, "Working in the GUI Map File per Test Mode."

At any stage in the testing process, you can switch from the *GUI Map File per Test* mode to the *Global GUI Map File* mode. For additional information, see Chapter 8, "Merging GUI Map Files."

As the user interface of your application changes, you can continue to use tests you developed previously. You simply add, delete, or edit object descriptions in the GUI map so that WinRunner can continue to find the objects in your modified application. For more information, see Chapter 7, "Editing the GUI Map."

You can specify which properties WinRunner uses to identify a specific class of object. You can also teach WinRunner to identify custom objects, and to map these objects to a standard class of objects. For additional information, see Chapter 9, "Configuring the GUI Map."

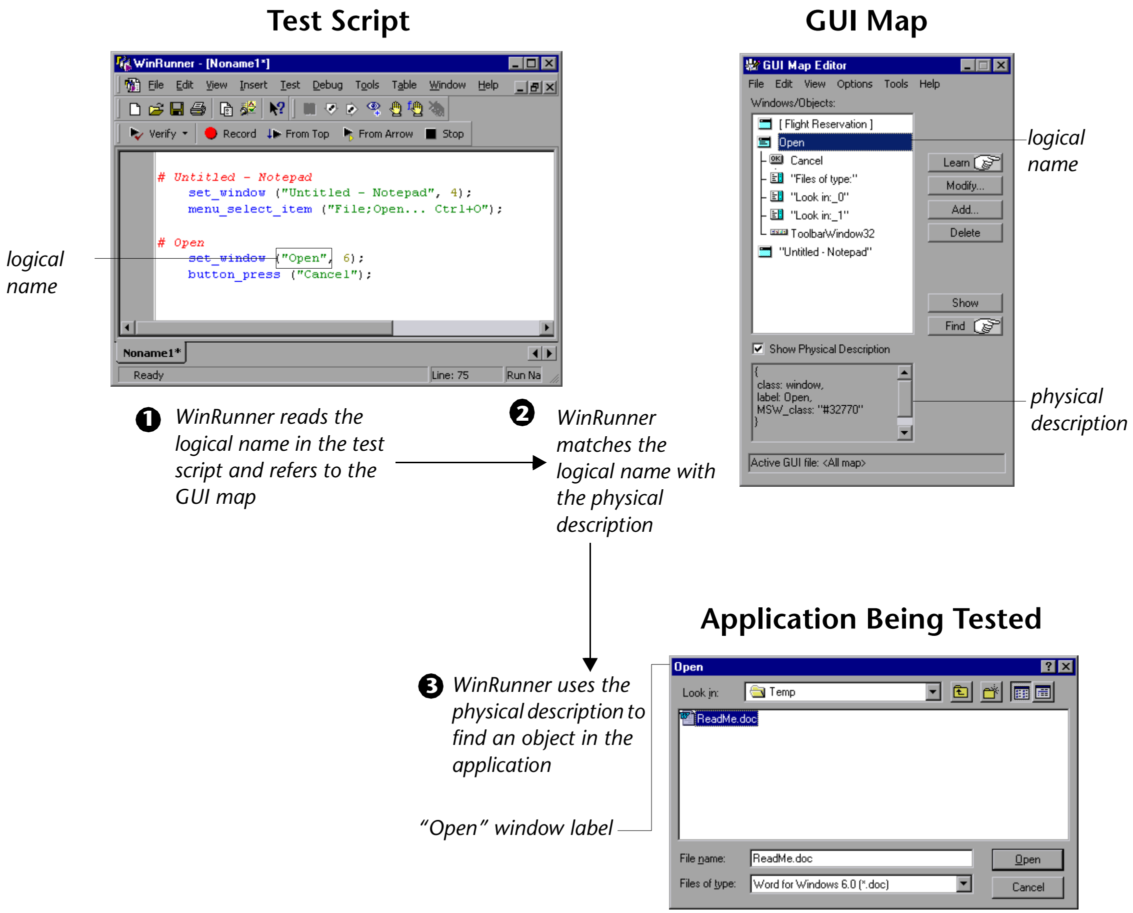
You can also teach WinRunner to recognize any bitmap in a window as a GUI object by defining the bitmap as a virtual object. For additional information, see Chapter 10, "Learning Virtual Objects."

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 11, "Designing Tests."

WinRunner uses a *logical name* to identify each object: for example "Print" for a Print dialog box, or "OK" for an OK button. The logical name is actually a nickname for the object's *physical description*. 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". The logical name and the physical description together ensure that each GUI object has its own unique identification.

Physical Descriptions



```
# Untitled - Notepad
    set_window ("Untitled - Notepad", 4);
    menu_select_item ("File;Open... Ctrl+O");

# Open
    set_window ("Open", 6);
    button_press ("Cancel");
```

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

☑ Show Physical Description

1. WinRunner reads the logical name in the test script and refers to the GUI map
2. WinRunner matches the logical name with the physical description
3. WinRunner uses the physical description to find an object in the application

WinRunner identifies each GUI object in the application being tested 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.

Each class has a set of default properties that WinRunner learns. For a detailed description of all properties, see Chapter 9, "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 32.

Note: Although WinRunner always identifies objects within the context of its window, a window's description is not dependent on the objects contained within it.

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)".

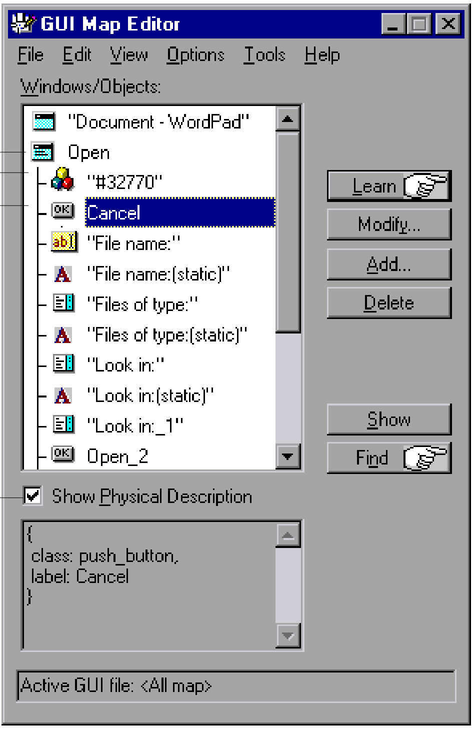
In certain cases, several GUI objects in the same window are assigned the same logical name, plus a location selector (for example: LogicalName_1, LogicalName_2). The purpose of the selector property is to create a unique name for the object.

The GUI Map

You can view the contents of the GUI map at any time by choosing **Tools** > **GUI Map Editor**. The GUI map is actually the sum of one or more GUI map files.

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.

For additional information on the GUI Map Editor, see Chapter 7, “Editing the GUI Map.”



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

Active GUI file: <All map>

This view shows the contents of the entire GUI map.

Window

Objects within the window

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

The GUI map file contains the logical names and physical descriptions of GUI objects.

There are two modes for organizing GUI map files:

➤ *Global GUI Map File* mode: You can create a GUI map file for your entire application, or for each window in your application. Different tests can reference a common GUI map file. For more information, see Chapter 5, “Working in the Global GUI Map File Mode.”

➤ *GUI Map File per Test* mode: WinRunner automatically creates a GUI map file that corresponds to each test you create. For more information, see Chapter 6, “Working in the GUI Map File per Test Mode.”

For a discussion of the relative advantages and disadvantages of each mode, see “Deciding Which GUI Map File Mode to Use” on page 43.

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.

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

4

Understanding Basic GUI Map Concepts

This chapter explains how WinRunner identifies the Graphical User Interface (GUI) of your application and how to work with GUI map files.

This chapter describes:

- About the GUI Map
- Viewing GUI Object Properties
- Teaching WinRunner the GUI of Your Application
- Finding an Object or Window in the GUI Map
- General Guidelines for Working with GUI Map Files
- Deciding Which GUI Map File Mode to Use

About the GUI Map

When WinRunner runs tests, it simulates a human user by moving the mouse cursor over the application, clicking GUI objects and entering keyboard input. Like a human user, WinRunner must learn the GUI of an application in order to work with it.

WinRunner does this by learning the GUI objects of an application and their properties and storing these object descriptions in the GUI map. You can use the GUI Spy to view the properties of any GUI object on your desktop, to see how WinRunner identifies it.

WinRunner can learn the GUI of your application in the following ways:

- by using the RapidTest Script wizard to learn the properties of all GUI objects in every window in your application
- by recording in your application to learn the properties of all GUI objects on which you record
- by using the GUI Map Editor to learn the properties of an individual GUI object, window, or all GUI objects in a window

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.

Before you start teaching WinRunner the GUI of your application, you should consider how you want to organize your GUI map files:

- In the *GUI Map File per Test* mode, WinRunner automatically creates a new GUI map file for every new test you create.
- In the *Global GUI Map File* mode, you can use a single GUI map for a group of tests.

The considerations for deciding which mode to use are discussed at the end of this chapter.

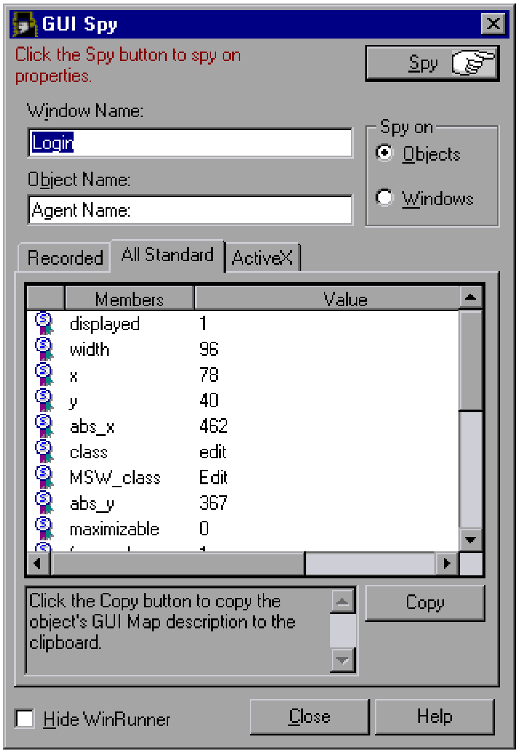
Viewing GUI Object Properties

When WinRunner learns the description of a GUI object, it looks at the object's physical properties. Each GUI object has many properties, such as "class," "label," "width," "height", "handle," and "enabled". WinRunner, however, learns only a selected set of these properties in order to uniquely distinguish the object from all other objects in the application.

Before you create the GUI map for an application, or before adding a GUI object to the GUI map, you may want to view the properties of the GUI object. Using the GUI Spy, you can view the properties of any GUI object on your desktop. You use the Spy pointer to 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 selected set of properties that WinRunner learns.

In the following example, pointing to the Agent Name edit box in the Login window of the sample flight application displays the **All Standard** tab in the GUI Spy as follows:



Click the Spy button to spy on properties.

Window Name: Login

Object Name: Agent Name:

Spy on: ● Objects ○ Windows

Members	Value
displayed	1
width	96
x	78
y	40
abs_x	462
class	edit
MSW_class	Edit
abs_y	367
maximizable	0

Click the Copy button to copy the object's GUI Map description to the clipboard.

□ Hide WinRunner

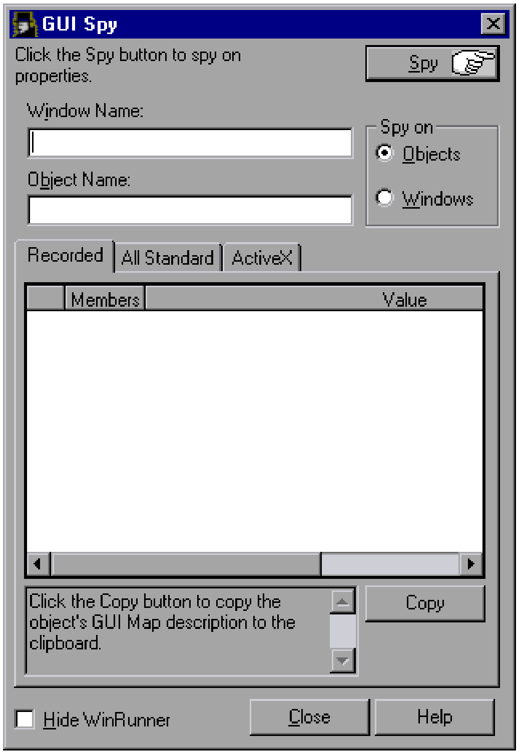
Tip: You can resize the GUI Spy to view the entire contents at once.

Note: The ActiveX tab is displayed only if the ActiveX Add-in is installed and loaded.

Note: You can modify the set of properties that WinRunner learns for a specific object class using the GUI Map Configuration dialog box. For more information on GUI Map Configuration, refer to Chapter 9, "Configuring the GUI Map."

To spy on a GUI object:

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



Click the Spy button to spy on properties.

Click the Copy button to copy the object's GUI Map description to the clipboard.

By default, the GUI Spy displays the Recorded tab, which enables you to view the properties of standard GUI objects that WinRunner records.

Tip: To view the properties of a window, click **Windows** in the **Spy on** box.

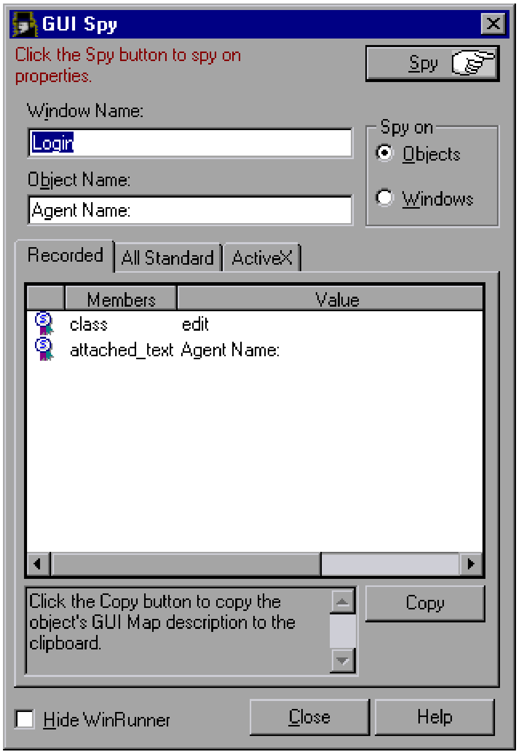
- To view all properties of standard windows and objects, click the **All Standard** tab.
- To view all properties and methods of ActiveX controls, click the **ActiveX** tab (only if the ActiveX Add-in is installed and loaded).

2. Select **Hide WinRunner** if you want to hide the WinRunner screen (but not the GUI Spy) while you spy on objects.

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.

In the following example, pointing to the Agent Name edit box in the Login window of the sample flight application displays the **Recorded** tab in the GUI Spy as follows:



Window Name: Login

Object Name: Agent Name:

Members	Value
class	edit
attached_text	Agent Name:

Click the Copy button to copy the object's GUI Map description to the clipboard.

4 To capture an object description in the GUI Spy dialog box, point to the desired object and press the STOP softkey. (The default softkey combination is CTRL LEFT + F3.)

If you selected **Hide WinRunner** before you began spying on objects, the WinRunner screen is displayed again when you press the STOP softkey.

➤ In the **Recorded** and **All Standard** tabs, you can click the **Copy** button to copy the physical description of the object to the Clipboard.

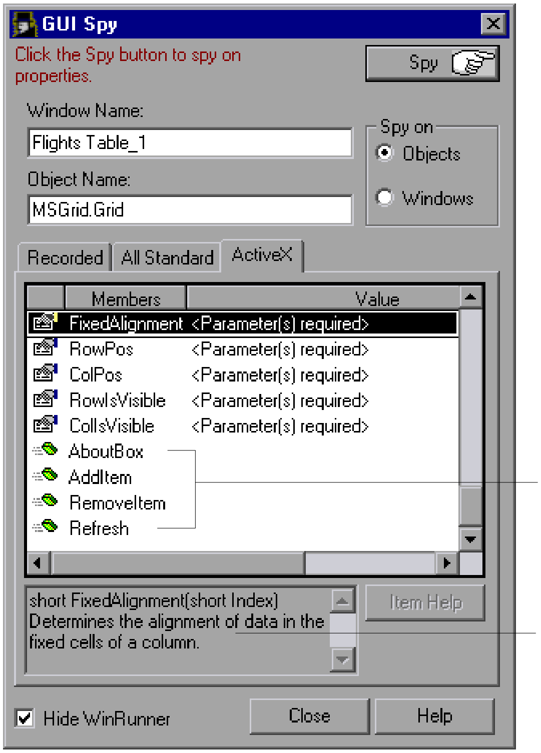
Clicking Copy in the previous example pastes the following physical description to the Clipboard:

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

Tip: You can press CTRL + C to copy the property and value from the selected row only to the Clipboard.

➤ When you highlight a property in the **ActiveX** tab, then if a description has been included for this property, it is displayed in the gray pane at the bottom. If a help file has been installed for this ActiveX control, then clicking **Item Help** displays it.

In the following example, pointing to the "Flights Table" in the Visual Basic sample flight application, pressing the STOP softkey and highlighting the FixedAlignment property, displays the **ActiveX** tab in the GUI Spy as follows:



Click the Spy button to spy on properties.

Window Name: Flights Table_1

Object Name: MSGrid.Grid

Spy on: ● Objects ○ Windows

Members	Value
FixedAlignment	<Parameter(s) required>
RowPos	<Parameter(s) required>
ColPos	<Parameter(s) required>
RowIsVisible	<Parameter(s) required>
ColIsVisible	<Parameter(s) required>
AboutBox	
AddItem	
RemoveItem	
Refresh	

short FixedAlignment(short Index)
Determines the alignment of data in the fixed cells of a column.

☑ Hide WinRunner

ActiveX control methods

property description

Note: If an ActiveX property value is a pointer (reference) to another object and that other object has a property marked by the control vendor as default then the GUI Spy shows a value of that default property rather than the value of the pointer. However, when using the **ActiveX_get_info** function for a property containing a pointer value, you should specify the property in the format **PropA.PropB**.

For example, if an ActiveX list object has a **SelectedItem** property, whose value is a pointer to another object representing the list item, and the list item's default property is the text property, then the GUI Spy will show the value of the text property, like ABC.

When using the **ActiveX_get_info** function:

```
ActiveX_get_info("LogName", "SelectedItem", RetVal)
```
returns a pointer value, like Object Reference - 0x782e789f.

```
ActiveX_get_info("LogName", "SelectedItem.Text", RetVal)
```
returns the text property value, like ABC.

5 Click **Close** to close the GUI Spy.

Teaching WinRunner the GUI of Your Application

Like a human user, WinRunner must learn the GUI of an application in order to work with it.

When you work in the *GUI Map File per Test* mode, you do not need to take any special steps to teach WinRunner the GUI of your application. WinRunner automatically learns the GUI of your application while you record.

When you work in the *Global GUI Map File* mode, you need to teach WinRunner the information it needs about the properties of GUI objects. WinRunner can learn this information in the following ways:

- using the RapidTest Script wizard to learn the properties of all GUI objects in every window in your application
- recording in your application to learn the properties of all GUI objects on which you record
- clicking the **Learn** button in the GUI Map Editor to learn the properties of an individual GUI object, window, or all GUI objects in a window

Note: When you work in the *GUI Map File per Test* mode, the RapidTest Script wizard is not available. The RapidTest Script wizard is also not available if the WebTest or certain other add-ins are loaded. To find out whether the RapidTest wizard is available with the add-in(s) you are using, refer to your add-in documentation.

For additional information on how to teach WinRunner the GUI of your application in the ways described above, see Chapter 5, “Working in the Global GUI Map File Mode.”

Finding an Object or Window in the GUI Map

When the cursor is on a statement in your test script that references a GUI object or window, you can right-click and select **Find in GUI Map**.

WinRunner finds and highlights the specified object or window in the GUI map or GUI map file and in the application, if it is open.

Note: Selecting GUI Maps or GUI Files on the View menu in the GUI Map Editor determines whether WinRunner locates the object in the GUI map or in a GUI map file.

- If the GUI map file containing the window is loaded, and the specified window is open, then WinRunner opens the GUI Map Editor and highlights the window in the GUI map and in the application.
- If the GUI map file containing the object is loaded, and the window containing the specified object is open, then WinRunner opens the GUI Map Editor and highlights the object in the GUI map and in the application.
- If the GUI map file containing the object or window is loaded, but the application containing the object or window is not open, then WinRunner opens the GUI Map Editor and highlights the object or window in the GUI map.

General Guidelines for Working with GUI Map Files

Consider the following guidelines when working with GUI map files:

- A single GUI map file cannot contain two windows with the same logical name.
- A single window in a GUI map file cannot contain two objects with the same logical name.
- In the GUI Map Editor, you can use the Options > Filter command to open the Filters dialog box and filter the objects in the GUI map by logical name, physical description, or class. For more information, see "Filtering Displayed Objects" on page 93.

Deciding Which GUI Map File Mode to Use

When you plan and create tests, you must consider how you want to work with GUI maps. You can work with one GUI map file for each test or a common GUI map file for multiple tests.

- If you are new to WinRunner or to testing, you may want to consider working in the *GUI Map File Per Test* mode. In this mode, a GUI map file is created automatically every time you create a new test. The GUI map file that corresponds to your test is automatically saved whenever you save your test and automatically loaded whenever you open your test.
- If you are familiar with WinRunner or with testing, it is probably most efficient to work in the *Global GUI Map File* mode. This is the default mode in WinRunner. All tests created in WinRunner 6.02 or lower were created in this mode. Note that whenever you work with a test created in WinRunner 6.02 or lower, you must work in this mode.

The following table lists the relative advantages and disadvantages of working in each mode:

	GUI Map File per Test	**Global GUI Map File**
Method	WinRunner learns the GUI of your application as you record and automatically saves this information in a GUI map file that corresponds to each test. When you open the test, WinRunner automatically loads the corresponding GUI map file.	Before you record, have WinRunner learn your application by clicking the **Learn** button in the GUI Map Editor and clicking your application window. You repeat this process for all windows in the application. You save the GUI map file for each window or set of windows as a separate GUI map file. When you run your test, you load the GUI map file. When the application changes, you update the GUI map files.
Advantages	1. Each test has its own GUI map file. 2. This is the simplest mode for inexperienced testers or WinRunner users who may forget to save or load GUI map files. 3. It is easy to maintain and update an individual test.	1. If an object or window description changes, you only have to modify one GUI map file for all tests referencing that file to run properly. 2. It is easy to maintain and update a suite of tests efficiently.

	GUI Map File per Test	Global GUI Map File
Disadvantages	Whenever the GUI of your application changes, you need to update the GUI map file for each test separately in order for your tests to run properly.	You need to remember to save and load the GUI map file, or to add statements that load the GUI map file to your startup test or to your other tests.
Recommendation	This is the preferred method if you are an inexperienced tester or WinRunner user or if the GUI of your application is not expected to change.	This is the preferred method for experienced WinRunner users and other experienced testers, or if the GUI of your application may change.

Note: Sometimes the logical name of an object is not descriptive. If you use the GUI Map Editor to learn your application before you record, then you can modify the name of the object in the GUI map to a descriptive name by highlighting the object and clicking the **Modify** button. When WinRunner records on your application, the new name will appear in the test script. For more information on modifying the logical name of an object, see “Modifying Logical Names and Physical Descriptions,” on page 81.

For additional guidelines on working in the Global GUI Map File mode, see “Guidelines for Working in the Global GUI Map File Mode” on page 66.

5

Working in the Global GUI Map File Mode

This chapter explains how to save the information in your GUI map when you work in the *Global GUI Map File* mode. This is the default mode in WinRunner. If you want to work in the simpler *GUI Map File per Test* mode, you can skip this chapter and proceed to Chapter 6, "Working in the GUI Map File per Test Mode."

Note: You must use this mode when working with tests created in WinRunner version 6.02 or earlier.

This chapter describes:

- About the Global GUI Map File Mode
- Sharing a GUI Map File among Tests
- Teaching WinRunner the GUI of Your Application
- Saving the GUI Map
- Loading the GUI Map File
- Guidelines for Working in the Global GUI Map File Mode

About the Global GUI Map File Mode

The most efficient way to work in WinRunner is to organize tests into groups when you design your test suite. Each test in the group should test the same GUI objects in your application. Therefore, these tests should reference the information about GUI objects in a common repository. When a GUI object in your application changes, you need to update the information only in the relevant GUI map file, instead of updating it in every test. When you work in the manner described above, you are working in the *Global GUI Map File* mode.

It is possible that one test within a test-group will test certain GUI objects within a window, while another test within the same group will test some of those objects and additional ones within the same window. Therefore, if you teach WinRunner the GUI of your application only by recording, your GUI map file may not contain a comprehensive list of all the objects in the window. It is best for WinRunner to learn the GUI of your application comprehensively before you start recording your tests.

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 use the GUI Map Editor to learn individual windows or objects. 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.

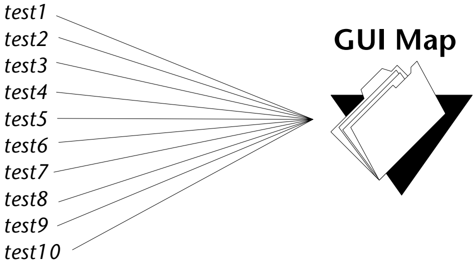
Note that since GUI map files are independent of tests, they are not saved automatically when you close a test. You must save the GUI map file whenever you modify it with changes you want to keep.

Similarly, since GUI map files are independent of tests, they are not automatically loaded when you open a test. Therefore, 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. It is most efficient to 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 46, "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.

Note: When you are working in the *Global GUI Map File* mode, then if you call a test created in the *GUI Map File per Test* mode that references GUI objects, the test may not run properly.

Sharing a GUI Map File among Tests

When you design your test suite so that a single GUI map file is shared by multiple tests, you can 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 relevant 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".

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 while it runs a test. See Chapter 7, "Editing the GUI Map," for more information.

Teaching WinRunner the GUI of Your Application

WinRunner must learn the information about the GUI objects in your application in order to add it to the GUI map file. WinRunner can learn the information it needs about the properties of GUI objects in the following ways:

- using the RapidTest Script wizard to teach WinRunner the properties of all GUI objects in every window in your application
- recording in your application to teach WinRunner the properties of all GUI objects on which you record
- using the GUI Map Editor to teach WinRunner the properties of an individual GUI object, window, or all GUI objects in a window

Teaching WinRunner the GUI with the RapidTest Script Wizard

You can use the RapidTest Script wizard before you start to test in order to teach WinRunner 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.

Note: You can use the RapidTest Script wizard only when you work in the *Global GUI Map File* mode (the default mode, which is described in this chapter). All tests created in WinRunner version 6.02 or earlier use this mode.

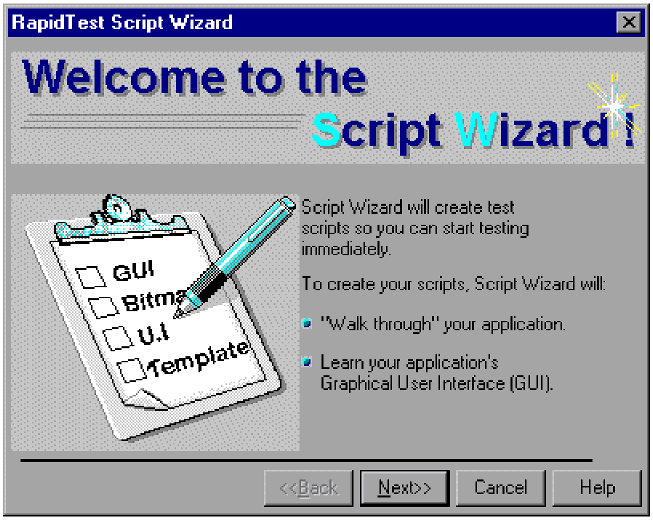
When you work in the *GUI Map File per Test* mode, the RapidTest Script wizard is not available. The RapidTest Script wizard is also not available if the WebTest or certain other add-ins are loaded. To find out whether the RapidTest wizard is available with the add-in(s) you are using, refer to your add-in documentation.

The simplest and most thorough way for WinRunner to learn your application is by using the RapidTest Script wizard. The RapidTest Script wizard enables WinRunner to learn all windows and objects in your application being tested at once. The wizard systematically opens each window in your application and learns the properties of the GUI objects it contains. WinRunner provides additional methods for learning the properties of individual objects.

WinRunner then saves the information in a GUI map file. WinRunner also creates a startup script which includes a **GUI_load** command that loads this GUI map file. For information on startup tests, see Chapter 46, “Initializing Special Configurations.”

To teach WinRunner your application using the RapidTest Script wizard:

1 Click **RapidTest Script Wizard** in the WinRunner Welcome screen when you start WinRunner or choose **Insert** > **RapidTest Script Wizard** at any time. The RapidTest Script wizard welcome screen opens.



RapidTest Script Wizard

Welcome to the Script Wizard !

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

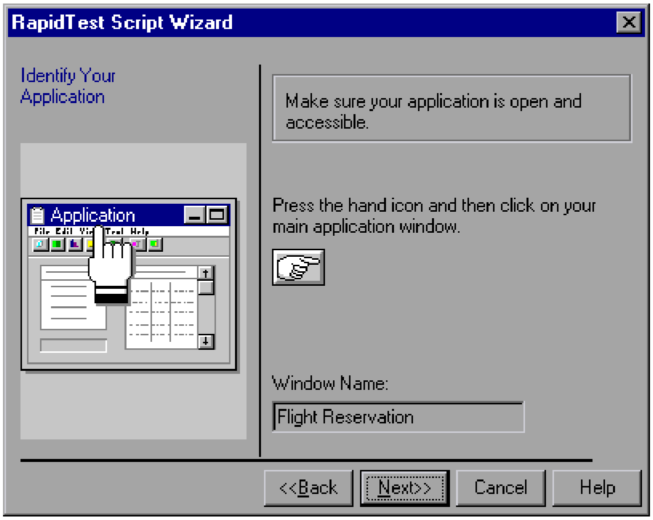
To create your scripts, Script Wizard will:

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

Click **Next**.

Note: The RapidTest Script Wizard option is not available when you use the WinRunner run-only version, when you work in *GUI file per test* mode, or when you load the WebTest add-in or certain other add-ins. Refer to your add-in documentation to see whether the RapidTest Script wizard is available when your add-in is loaded.

2 The **Identify Your Application** screen opens.



RapidTest Script Wizard

Identify Your Application

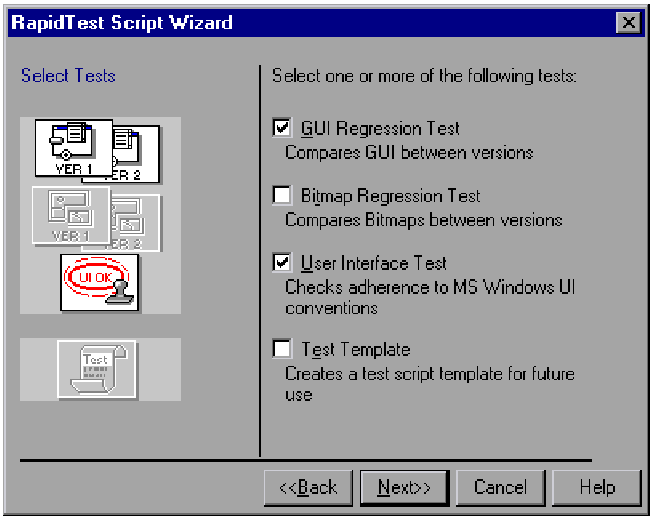
Make sure your application is open and accessible.

Press the hand icon and then click on your main application window.

Window Name: Flight Reservation

Click the pointing hand, and then click your application in order to identify it for the Script wizard. The name of the window you clicked appears in the Window Name box. Click **Next**.

3 The **Select Tests** screen opens.



RapidTest Script Wizard

Select Tests

Select one or more of the following tests:

☑ GUI Regression Test
Compares GUI between versions

☐ Bitmap Regression Test
Compares Bitmaps between versions

☑ User Interface Test
Checks adherence to MS Windows UI conventions

☐ Test Template
Creates a test script template for future use

4 Select the type(s) of test(s) you want WinRunner to create for you. When the Script Wizard finishes walking through your application, the tests you select are displayed in the WinRunner window.

You can choose any of the following tests:

- **GUI Regression Test -** This test enables you to compare the state of GUI objects in different versions of your application. For example, it can check whether a button is enabled or disabled.

  To create a GUI Regression test, the wizard captures default information about each GUI object in your application. When you run the test on your application, WinRunner compares the captured state of GUI objects to their current state, and reports any mismatches.

- **Bitmap Regression Test -** This test enables you to compare bitmap images of your application in different versions of your application. Select this test if you are testing an application that does not contain GUI objects.

  To create a Bitmap Regression test, the wizard captures a bitmap image of each window in your application. When you run the test, WinRunner compares the captured window images to the current windows, and reports any mismatches.

- **User Interface Test -** This test determines whether your application adheres to Microsoft Windows standards. It checks that:

  - GUI objects are aligned in windows
  - All defined text is visible on a GUI object
  - Labels on GUI objects are capitalized
  - Each label includes an underlined letter (mnemonics)
  - Each window includes an **OK** button, a **Cancel** button, and a system menu

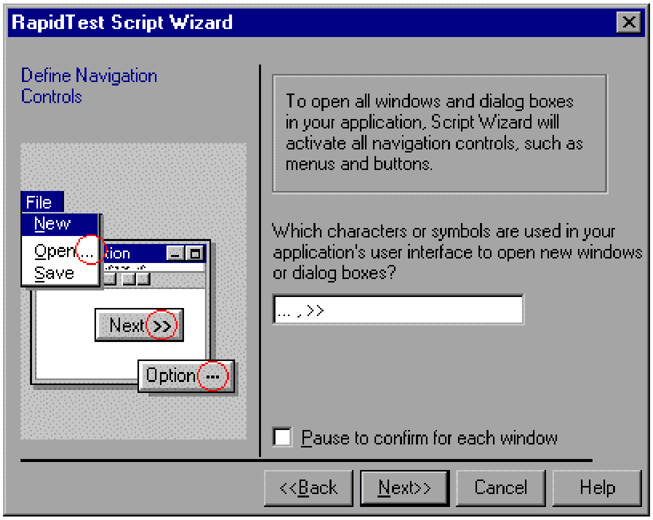
  When you run this test, WinRunner searches the user interface of your application and reports each case that does not adhere to Microsoft Windows standards.

- **Test Template -** This test provides a basic framework of an automated test that navigates your application. It opens and closes each window, leaving space for you to add code (through recording or programming) that checks the window.

Tip: Even if you do not want to create any of the tests described above, you can still use the Script wizard to learn the GUI of your application.

Click **Next**.

5 The **Define Navigation Controls** screen opens.



RapidTest Script Wizard

Define Navigation Controls

To open all windows and dialog boxes in your application, Script Wizard will activate all navigation controls, such as menus and buttons.

Which characters or symbols are used in your application's user interface to open new windows or dialog boxes?

... , >>

□ Pause to confirm for each window

Enter the characters that represent navigation controls in your application. If you want the RapidTest Script wizard to pause in each window in your application, so that you can confirm which objects will be activated to open additional windows, select the **Pause to confirm for each window** check box.

Click **Next**.

6. Choose **Express** or **Comprehensive** learning flow. Click **Learn**. WinRunner begins to systematically learn your application, one window at a time. This may take several minutes depending on the complexity of your application.

7. Choose **Yes** or **No** to tell WinRunner whether or not you want WinRunner to automatically activate this application whenever you invoke WinRunner.

   Click **Next**.

8. Enter the full path and file name where you want your startup script and GUI Map file to be stored, or accept the defaults.

   Click **Next**.

9. Enter the full path and file name where you want your test files to be stored, or accept the defaults.

   Click **Next**.

10. Click **OK** to close the RapidTest Script wizard. The test(s) that were created based on the application that WinRunner learned are displayed in the WinRunner window.

Teaching WinRunner the GUI by Recording

WinRunner can also learn objects while recording in Context Sensitive mode (the default mode) in your application: 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 or the GUI Map Editor if you plan to develop comprehensive test suites. For information on recording in Context Sensitive mode, see Chapter 11, “Designing Tests.”

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.

WinRunner adds the information it learned to the temporary GUI map file. To save the information in the temporary GUI map file, you must save this file before exiting WinRunner. For additional information on saving the GUI map, see “Saving the GUI Map” on page 59.

Tip: If you do not want WinRunner to add the information it learns to the temporary GUI map file, you can instruct WinRunner not to load the temporary GUI map file in the **General** category of the General Options dialog box. For more information, see Chapter 41, “Setting Global Testing Options.”

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

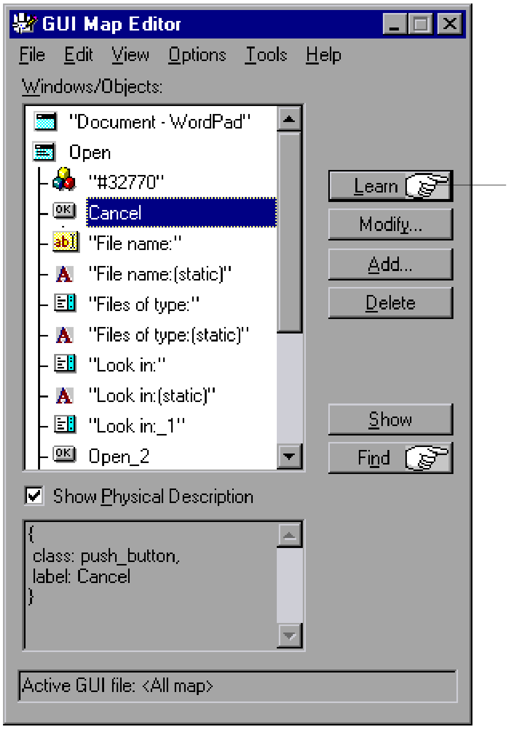
Teaching WinRunner the GUI Using the GUI Map Editor

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

To teach GUI objects to WinRunner using the GUI Map Editor:

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

Click **Learn**. The mouse pointer becomes a pointing hand.



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

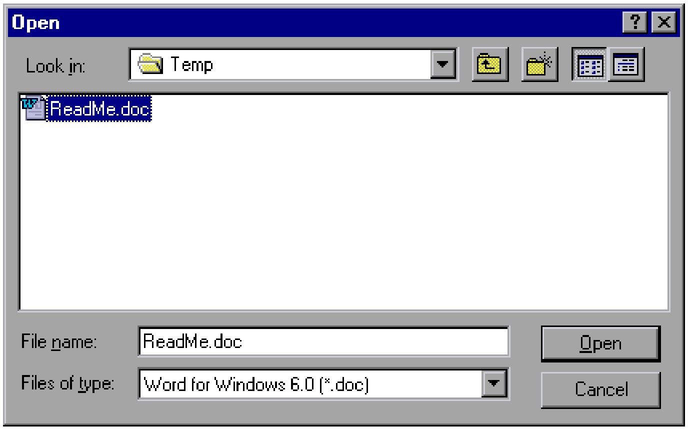
Learns the objects in a window.

- To learn all the objects in a window, click the title bar of the window. When prompted to learn all the objects in the window, click **Yes** (the default).
- To learn only a window, click the title bar of the window. When prompted to learn all the objects in the window, click **No**.
- To learn an object, click the object.

(To cancel the operation, click the right mouse button.)

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 the left mouse button.



Look in: Temp

File name: ReadMe.doc

Files of type: Word for Windows 6.0 (*.doc)

WinRunner adds the information it learns to the temporary GUI map file. To keep the information in the temporary GUI map file, you must save it before exiting WinRunner. For additional information on saving the GUI map, see Chapter 5, "Working in the Global GUI Map File Mode."

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.

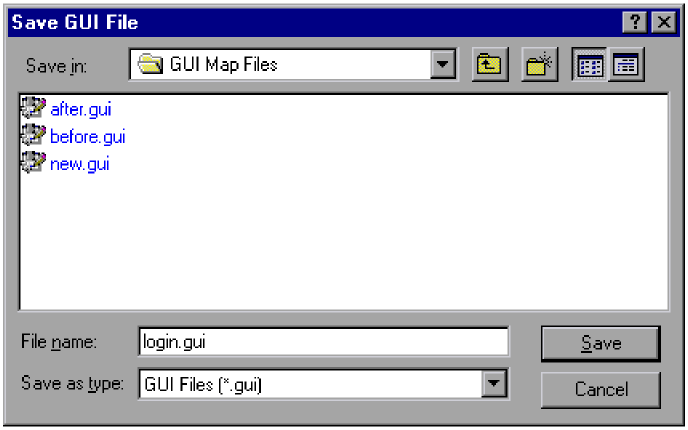
To avoid overwriting valuable GUI information during a new recording session, you should save the temporary GUI map file in a permanent GUI map file.

To save the contents of the temporary GUI map file to 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.



File name: login.gui

Save as type: GUI Files (*.gui)

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 "Copying and Moving Objects between Files" on page 88.

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

Note: You can only save GUI map files to a TestDirector database if you are working with TestDirector. For additional information, see Chapter 48, "Managing the Testing Process."

---

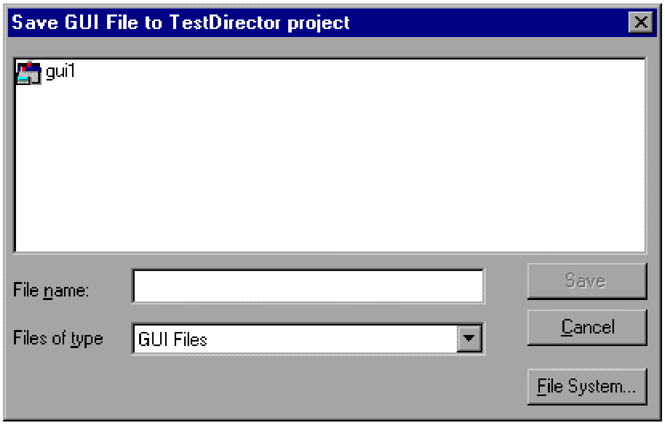
1 Choose **Tools** > **GUI Map Editor** to open the GUI Map Editor.

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**.

The Save GUI File to TestDirector project dialog box opens.



Save GUI File to TestDirector project

gui1

File name:

Files of type: GUI Files

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.

Loading the GUI Map File

When WinRunner learns the objects in an application, it stores the information in a GUI map file. In order for WinRunner to use a GUI map file to locate objects in your application, you must *load* it into the GUI map. 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. You can also open the GUI map file for editing without loading it.

Note: If you are working in the *GUI Map File per Test* mode, you should not manually load, unload, or save GUI map files.

Loading GUI Map Files Using the GUI_load Function

The **GUI_load** statement loads any GUI map file you specify. Although the GUI map may contain one or more GUI map files, you can load only one GUI map file at a time. 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. In this way, GUI map files are loaded automatically each time you start WinRunner. For more information, see Chapter 46, "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 browse to or type in the file path:

   GUI_load ("*file_name_full_path*"**);**

   For example:

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

   See Chapter 27, "Generating Functions," for information on how to use the Function Generator.

3 Run the test to load the file. See Chapter 33, "Understanding Test Runs," for more information.

Note: If you only want to edit the GUI map file, you can use the **GUI_open** function to open a GUI map file for editing, without loading it. You can use the **GUI_close** function to close an open GUI map file. See Chapter 7, "Editing the GUI Map," for information about editing the GUI map file. You can use the **GUI_unload** and **GUI_unload_all** functions to unload loaded GUI map files. For information on working with TSL functions, see Chapter 26, "Enhancing Your Test Scripts with Programming." For more information about specific TSL functions and examples of usage, refer to the *TSL Reference*.

Loading GUI Map Files Using the GUI Map Editor

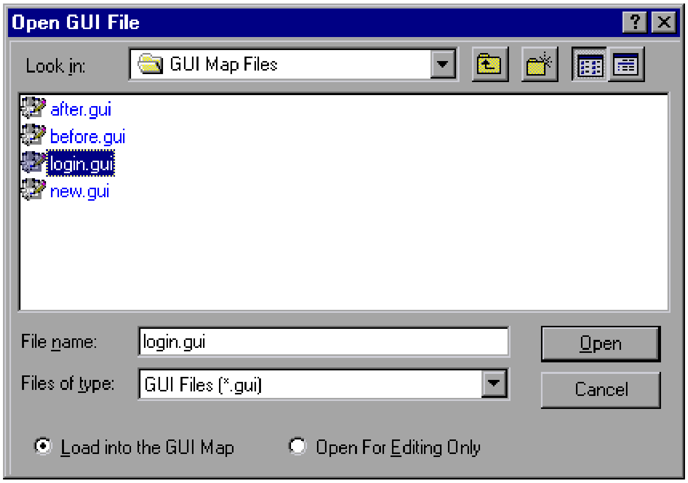
You can load a GUI map file manually from the file system or from a TestDirector database, using the GUI Map Editor.

Note: You can only load GUI map files from a TestDirector database if you are connected to a TestDirector project. For additional information, see Chapter 48, "Managing the Testing Process."

To load a GUI map file from the file system 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.



Open GUI File

Look in: GUI Map Files

after.gui
before.gui
login.gui
new.gui

File name: login.gui

Files of type: GUI Files (*.gui)

● Load into the GUI Map ○ Open For Editing Only

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 7, "Editing the GUI Map," for information about editing the GUI map file.

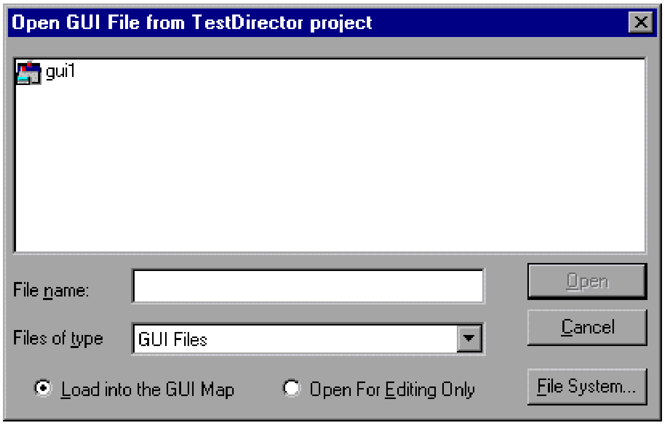
5 Click **Open**. The GUI map file is added to the GUI file list. The letter "L" and a number preceding the file name indicates that the file has been loaded.

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 selected database are listed in the dialog box.



3 Select a GUI map file from the list of GUI map files in the selected database. The name of the GUI map file appears in the **File name** text box.

To load the GUI map file into the GUI Map Editor, make sure the **Load into the GUI Map** default setting is selected. Alternatively, if you only want to edit the GUI map file, click **Open For Editing Only**. For more information, see Chapter 7, “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.

Guidelines for Working in the Global GUI Map File Mode

Consider the following guidelines when working in the *Global GUI Map File* mode:

- To improve performance, use smaller GUI map files for testing your application instead of one larger file. You can divide your application's user interface into different GUI map files by window or in another logical manner.
- Sometimes the logical name of an object is not descriptive. If you use the GUI Map Editor to learn your application before you record, then you can modify the logical name of the object in the GUI map to a descriptive name by highlighting the object and clicking the **Modify** button. When WinRunner records on your application, the new name will appear in the test script. If you recorded your test before changing the logical name of the object in the GUI map, make sure to update the logical name of the object accordingly in your test script before you run your test. For more information on modifying the logical name of an object, see "Modifying Logical Names and Physical Descriptions," on page 81.
- Do not store information that WinRunner learns about the GUI of an application in the temporary GUI map file, since this information is not automatically saved when you close WinRunner. Unless you are creating a small, temporary test that you do not intend to reuse, you should save the GUI map from the GUI Map Editor (by choosing **File** > **Save**) before closing your test.

Tip: You can instruct WinRunner not to load the temporary GUI map file in the **General** category of the General Options dialog box. For more information on this option, see Chapter 41, "Setting Global Testing Options."

- When WinRunner learns the GUI of your application by recording, it learns only those objects upon which you perform operations; it does not learn all the objects in your application. Therefore, unless you are creating a small, temporary test that you do not intend to reuse, it is best for WinRunner to learn the GUI of an application from the RapidTest Script wizard or the **Learn** button in the GUI Map Editor before you start recording than for WinRunner to learn your application once you start recording.
- Consider appointing one tester a “GUI Map Administrator,” with responsibility for updating the GUI maps when the GUI of your application changes.

For additional guidelines for working with GUI maps, see “General Guidelines for Working with GUI Map Files” on page 43.

6

Working in the GUI Map File per Test Mode

This chapter explains how to work in the *GUI Map File per Test* mode. This mode is recommended if you are new to testing or to WinRunner. It is very easy to use because you do not need to understand how to create, save, or load GUI map files.

Note: This feature is available for tests created in WinRunner version 7.0 or later only. You cannot use this mode for tests created in WinRunner version 6.02 or earlier.

This chapter describes:

- About the GUI Map File per Test Mode
- Specifying the GUI Map File per Test Mode
- Working in the GUI Map File per Test Mode
- Guidelines for Working in the GUI Map File per Test Mode

About the GUI Map File per Test Mode

When you work in the *GUI Map File per Test* mode, you do not need to teach WinRunner the GUI of your application, save, or load GUI map files (as discussed in Chapter 5, “Working in the Global GUI Map File Mode”), since WinRunner does this for you automatically.

In the *GUI Map File per Test* mode, WinRunner creates a new GUI map file whenever you create a new test. WinRunner saves the test's GUI map file whenever you save the test. When you open the test, WinRunner automatically loads the GUI map file associated with the test.

Note that some WinRunner features are not available when you work in this mode:

- The RapidTest Script wizard is disabled. For information about this wizard, see Chapter 5, "Working in the Global GUI Map File Mode."
- The option to reload the (last) temporary GUI map file when starting WinRunner (the **Load temporary GUI map file** check box in the **General category** of the General Options dialog box) is disabled. For additional information about this option, see Chapter 41, "Setting Global Testing Options."
- The *myinit* startup test that loads GUI map files when starting WinRunner does not load GUI map files. For information about startup tests, see Chapter 46, "Initializing Special Configurations."
- Compiled modules do not load GUI map files. If a compiled module references GUI objects, then those objects must also be referenced in the test that loads the compiled module. For additional information, see Chapter 30, "Creating Compiled Modules."
- If a called test that was created in the *GUI Map File per Test* mode references GUI objects, it may not run properly in the *Global GUI Map File* mode.

You choose to work in the *GUI Map File per Test* mode by specifying this option in the **General** category of the General Options dialog box.

When you become more familiar with WinRunner, you may want to consider working in the *Global GUI Map File* mode. In order to change from working in the *GUI Map File per Test* mode to working in the *Global GUI Map File* mode, you must merge the GUI map files associated with each test into GUI map files that are common to a test-group. You can use the GUI Map File Merge Tool to merge GUI map files. For additional information on merging GUI map files and changing to the *Global GUI Map File* mode, see Chapter 8, "Merging GUI Map Files."

Specifying the GUI Map File per Test Mode

In order to work in the *GUI Map File per Test* mode, you must specify this option in the **General** category of the General Options dialog box.

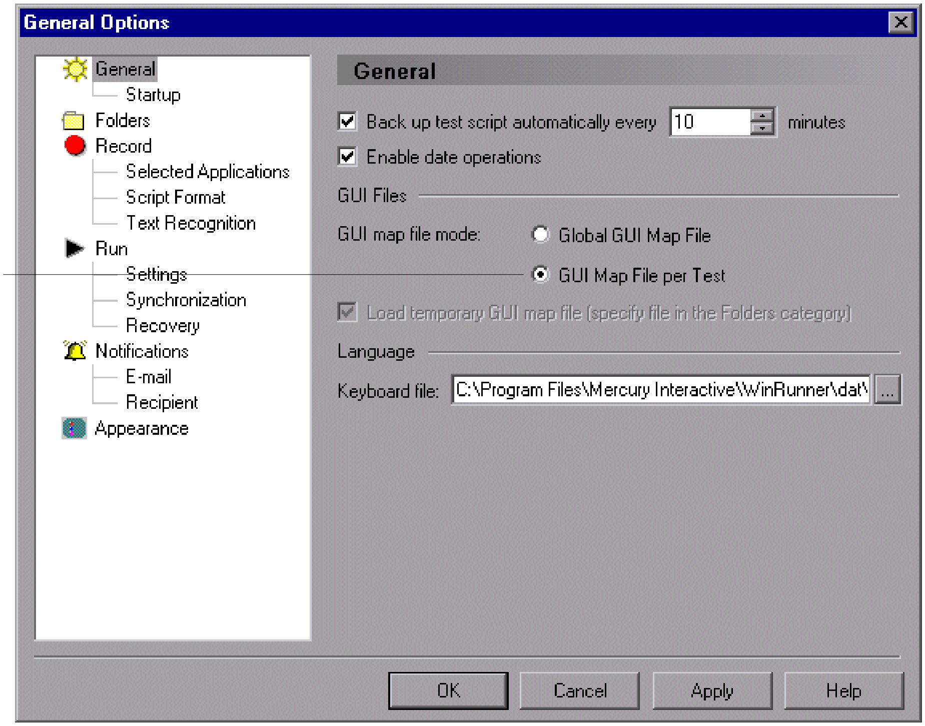
To work in the *GUI Map File per Test* mode:

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

The General Options dialog box opens.

2 Click the **General** category.

3 In the GUI files section, select **GUI Map File per Test**.



Set the GUI Map File per Test mode.

General Options

General

☑ Back up test script automatically every 10 minutes

☑ Enable date operations

GUI Files

GUI map file mode: ○ Global GUI Map File ● GUI Map File per Test

☑ Load temporary GUI map file (specify file in the Folders category)

Language

Keyboard file: C:\Program Files\Mercury Interactive\WinRunner\dat\

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

Note that the **Load temporary GUI map file** option is automatically disabled.

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

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

For additional information on the General Options dialog box, see Chapter 41, "Setting Global Testing Options."

Working in the GUI Map File per Test Mode

Every time you create a new test, WinRunner automatically creates a new GUI map file for the test. Whenever you save the test, WinRunner saves the corresponding GUI map file. The GUI map file is saved in the same folder as the test. Moving a test to a new location also moves the GUI map file associated with the test.

WinRunner learns the GUI of your application by recording. If the GUI of your application changes, you can update the GUI map file for each test using the GUI Map Editor. You do not need to load or save the GUI map file.

To update a GUI map file:

1. Open the test for which you want to update the GUI map file.
2. Choose **Tools** > **GUI Map Editor** to open the GUI Map Editor.
3. Edit the open GUI map file, as described in Chapter 7, "Editing the GUI Map."

Note: If you change the logical name of an object in your GUI map file, you must update your test script accordingly. For additional information, see "Modifying Logical Names and Physical Descriptions" on page 81.

4. When you are done, choose **File** > **Exit** to close the GUI Map Editor.

Guidelines for Working in the GUI Map File per Test Mode

Consider the following guidelines when working in the *GUI Map File per Test* mode:

- Do not save your changes to a GUI map file from the GUI Map Editor. Your changes are saved automatically when you save your test.
- Do not manually load or unload GUI map files while working in the *GUI Map File per Test* mode. The GUI map file for each test is automatically loaded when you open your test.

For additional guidelines for working with GUI maps, see "General Guidelines for Working with GUI Map Files" on page 43.

7

Editing the GUI Map

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

This chapter describes:

About Editing the GUI Map

WinRunner uses the GUI map to identify and locate GUI objects in your application. If the GUI of your application changes, you must update object descriptions in the GUI map so 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 5, "Working in the Global GUI Map File Mode," for more information.

Note: If you are working in the *GUI Map File per Test* mode, you should not manually load or unload GUI map files.

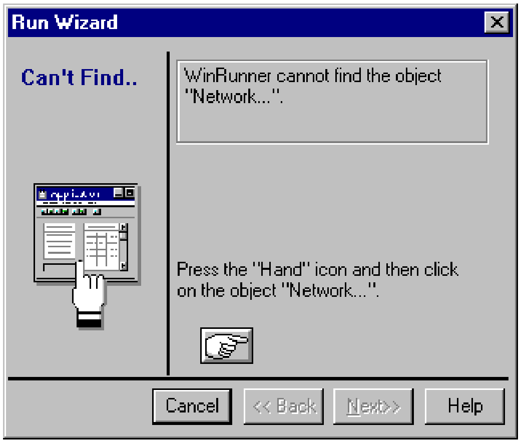
The Run Wizard

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 opens when WinRunner cannot locate an object. The Run wizard prompts you to point to the object in your application, determines why the object cannot be found, and then offers a solution. For example, the Run wizard may suggest 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. This portion of your script may appear as follows:

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

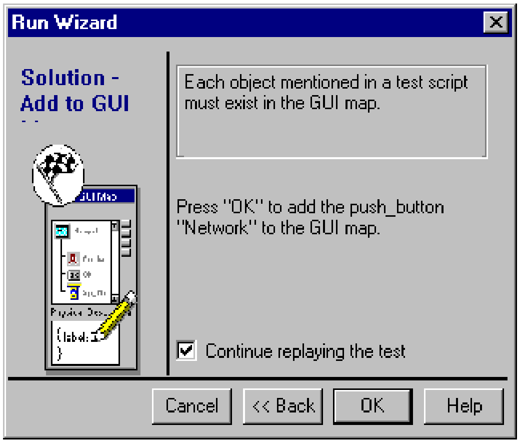
If the Network button is not in the GUI map, the Run wizard opens and describes the problem.



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

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

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



Run Wizard

Solution - Add to GUI

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

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

☑ Continue replaying the test

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

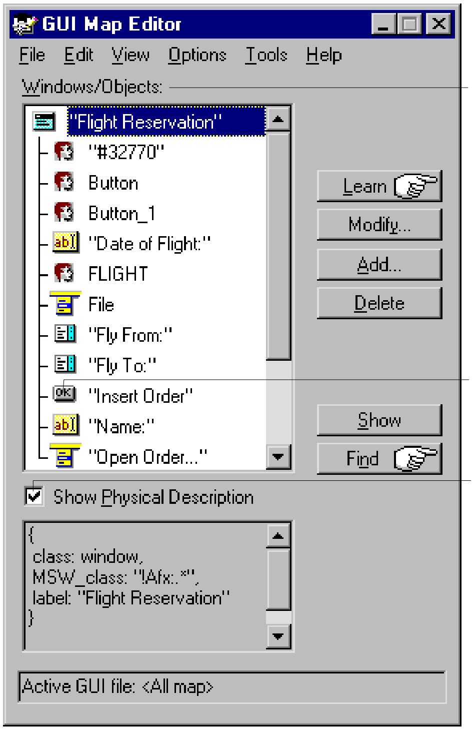
In some cases, the Run wizard edits the test script, rather than the GUI map. For example, if WinRunner cannot locate an object because the appropriate window is inactive, the Run wizard inserts a **set_window** statement in the test script.

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**.

There are two views in the GUI Map Editor, which enable you to display the contents of either:

- the entire GUI map
- an individual GUI map file



"Flight Reservation"
"#32770"
Button
Button_1
"Date of Flight:"
FLIGHT
File
"Fly From:"
"Fly To:"
"Insert Order"
"Name:"
"Open Order..."

☑ Show Physical Description

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

Active GUI file: <All map>

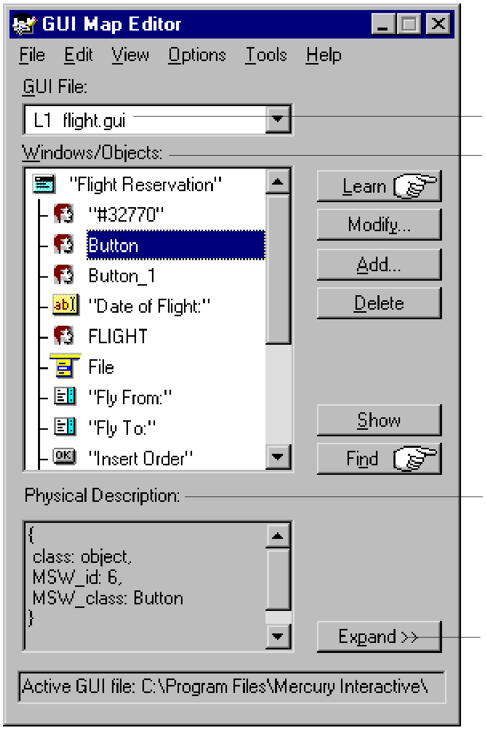
Displays all windows and objects in the GUI map.

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**.



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

Lists the open GUI map files.

Shows the windows and objects in the selected 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 you modify the logical name of an object in the GUI map, you must also modify the logical name of the object in the test script, so that WinRunner will be able to locate the object in the GUI map.

Note: If the value of a property contains any spaces or special characters, that value must be enclosed by quotation marks. Multiple property:value sets must be separated by commas.

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:

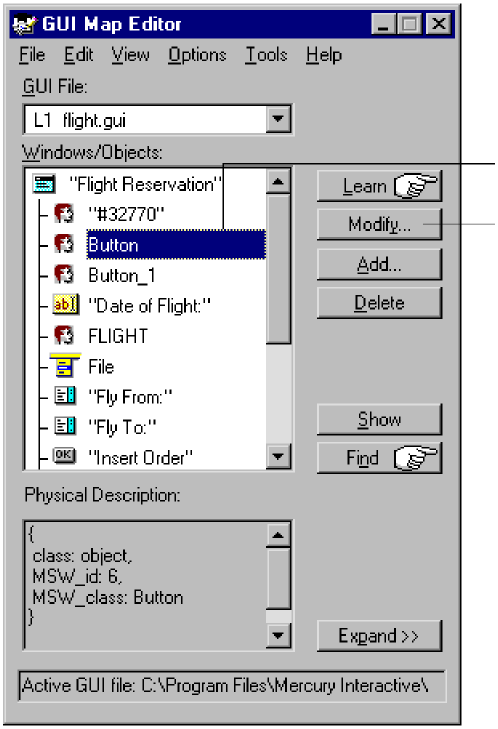
```
Insert button:{class:push_button, label:Add}
```

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

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 already 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.

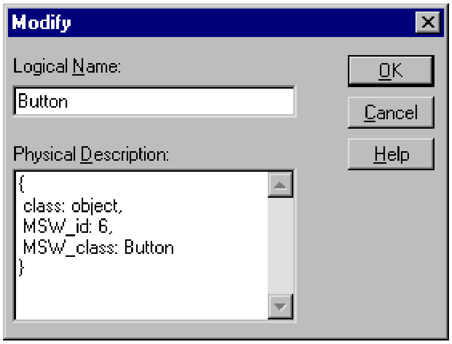


```
{
 class: object,
 MSW_id: 6,
 MSW_class: Button
}
```

Select a window or an object.

Click Modify.

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



Logical Name: Button

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

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

Adding Comments to the Physical Description

When you modify an object's physical description, you can add comments to make the physical description easier to understand. For example, suppose you want to add a comment that makes it easier for you to recognize the object. You could write:

```
{
 class: object,
 MSW_class: html_text_link,
 html_name: here,
 comment: "Link to the home page"
}
```

Note: As with any other property, if the value of a comment property contains any spaces or special characters, that value must be enclosed by quotation marks.

How WinRunner Handles Varying Window Labels

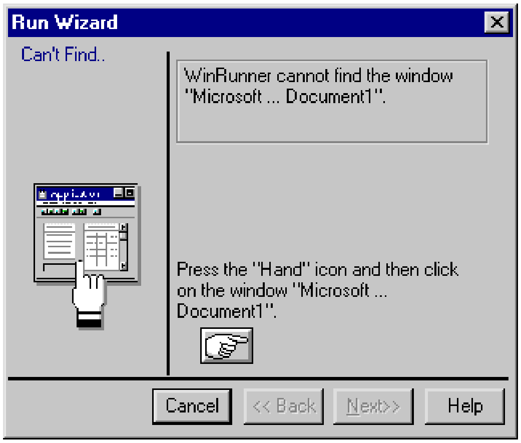
Windows often have varying labels. For example, the main window in a text application might display a file name as well as the application name in the title bar.

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.

For example, suppose you record a test on the main window of Microsoft Word. WinRunner learns the following physical description:

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

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



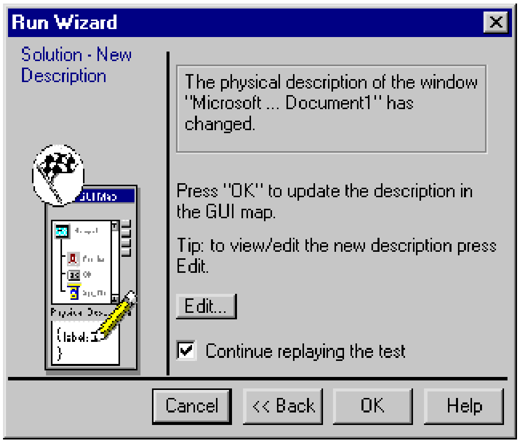
Run Wizard

Can't Find..

WinRunner cannot find the window "Microsoft ... Document1".

Press the "Hand" icon and then click on the window "Microsoft ... Document1".

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.



The physical description of the window "Microsoft ... Document1" has changed.

Press "OK" to update the description in the GUI map.

Tip: to view/edit the new description press Edit.

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 "- Document" window title.

Using 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.

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 to the GUI configuration for a class as needed. 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 9, “Configuring the GUI Map,” for more information.

For more information about regular expressions, see Chapter 25, “Using Regular Expressions.”

Copying and Moving Objects between Files

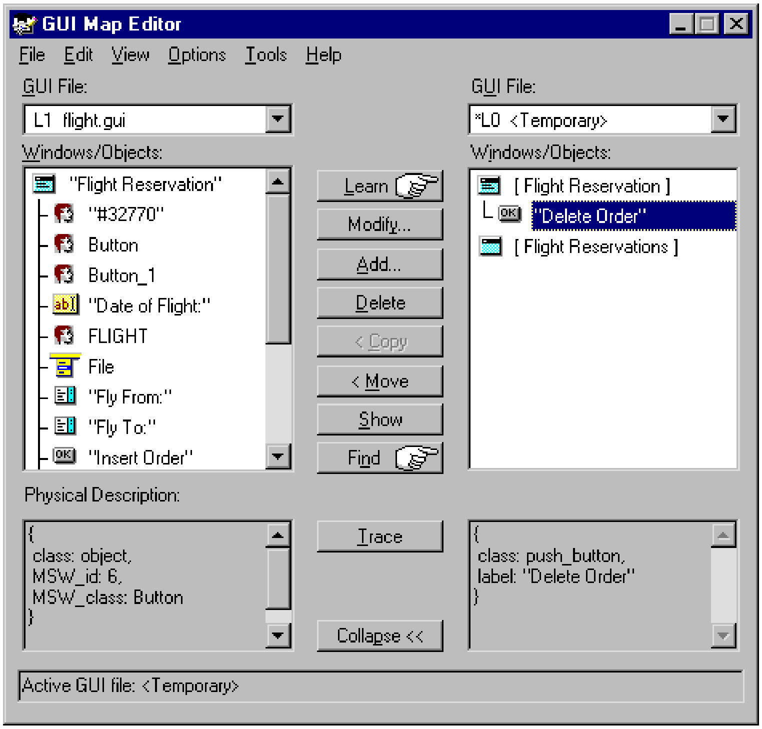
You can update GUI map files by copying or moving the description of GUI objects from one GUI map file to another. Note that you can only copy objects from a GUI file that you have opened for editing only, that is, from a file you have not loaded.

Note: If you are working in the *GUI Map File per Test* mode, you should not manually open GUI map files or copy or move objects between files.

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.



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

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

Active GUI file: <Temporary>

4 View a different GUI map file on each side of the dialog box by selecting 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 GUI map file, choose **Edit** > **Select All**.

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

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

Note: If you add new windows from a loaded GUI map file to the temporary GUI map file, then when you save 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 Help*.

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.

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.

Select 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** box.

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.

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:

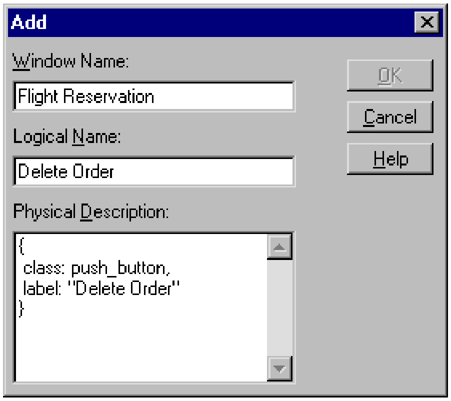
1 Choose **Tools** > **GUI Map Editor** to open the GUI Map Editor.

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.



Window Name: Flight Reservation

Logical Name: Delete Order

Physical Description:

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

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

Deleting an Object from a GUI Map File

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

To delete an object from a GUI map file:

1 Choose **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 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. 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**.

Filtering Displayed Objects

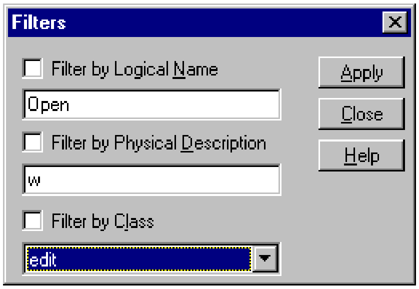
You can filter the list of objects displayed in the GUI Map Editor by using any of the following filters:

- *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” displays only objects containing a “w” in their physical description.)
- *Class* displays only objects of the specified class, such as all the push buttons.

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.



☐ Filter by Logical Name

Open

☐ Filter by Physical Description

w

☐ Filter by Class

edit

3 Select the type of filter you want by selecting a check box and entering the appropriate information.

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

Saving Changes to the GUI Map

If you edit the logical names and physical descriptions of objects in the GUI map or modified the objects or windows within a GUI map file, you must save your changes in the GUI Map Editor before ending the testing session and exiting WinRunner.

Note: If you are working in the *GUI Map File per Test* mode, you should not manually save changes to the GUI map. Your changes are saved automatically with your test.

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.

Note: If you add new windows from a loaded GUI map file to the temporary GUI map file, then when you save 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 Help*.

8

Merging GUI Map Files

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

This chapter describes:

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

About Merging GUI Map Files

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

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

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

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

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

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

Preparing to Merge GUI Map Files

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

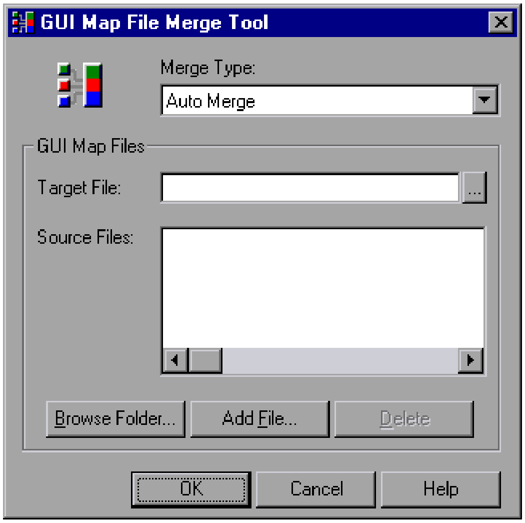
To start merging GUI map files:

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

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

- To continue, click **OK**.
- To save changes to open GUI maps, click **Cancel** and save the GUI maps using the GUI Map Editor. For information on saving GUI map files, see “Saving Changes to the GUI Map” on page 94. Once you have saved changes to the open GUI map files, start again at step 1.

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



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

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

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

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

4 Specify the source GUI map files.

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

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

- ➤ If you chose **Auto Merge** and the source GUI map files are merged successfully without conflicts, a message confirms the merge.
- ➤ If you chose **Auto Merge** and there are conflicts among the source GUI map files being merged, a WinRunner message box warns of the problem. When you click OK to close the message box, the GUI Map File Auto Merge Tool opens. For additional information, see "Resolving Conflicts while Automatically Merging GUI Map Files" on page 100.
- ➤ If you chose **Manual Merge**, the GUI Map File Manual Merge Tool opens. For additional information, see "Merging GUI Map Files Manually" on page 103.

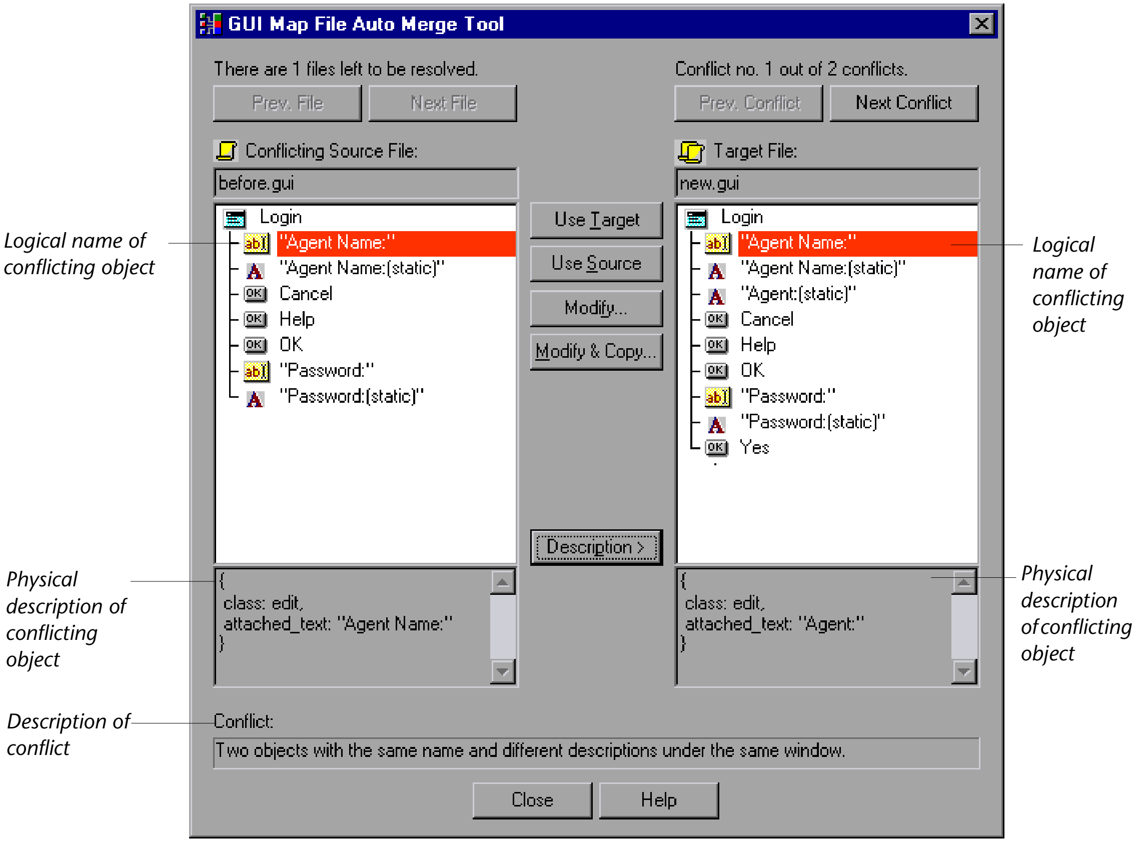
Resolving Conflicts while Automatically Merging GUI Map Files

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

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

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

The following example demonstrates automatically merging two conflicting source files (*before.gui* and *after.gui*) into a new target file (*new.gui*). The GUI Map File Auto Merge Tool opens after clicking OK in the conflict warning message box, as described in “Preparing to Merge GUI Map Files” on page 98. It enables you to resolve conflicts and prevents you from creating new conflicts in the target file.



GUI Map File Auto Merge Tool

There are 1 files left to be resolved.

Conflict no. 1 out of 2 conflicts.

Conflicting Source File: before.gui

Target File: new.gui

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

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

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

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

The source files are merged in the order in which they appear in the GUI Map File Merge Tool, as described in "Preparing to Merge GUI Map Files" on page 98.

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

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

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

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

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

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

Tip: Sometimes, all the conflicts in the current source file will have been resolved as a result of resolving conflicts in other source files. When this happens, the **Remove File** button is displayed. Click this button to remove the current source file from the list of source GUI map files.

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

Merging GUI Map Files Manually

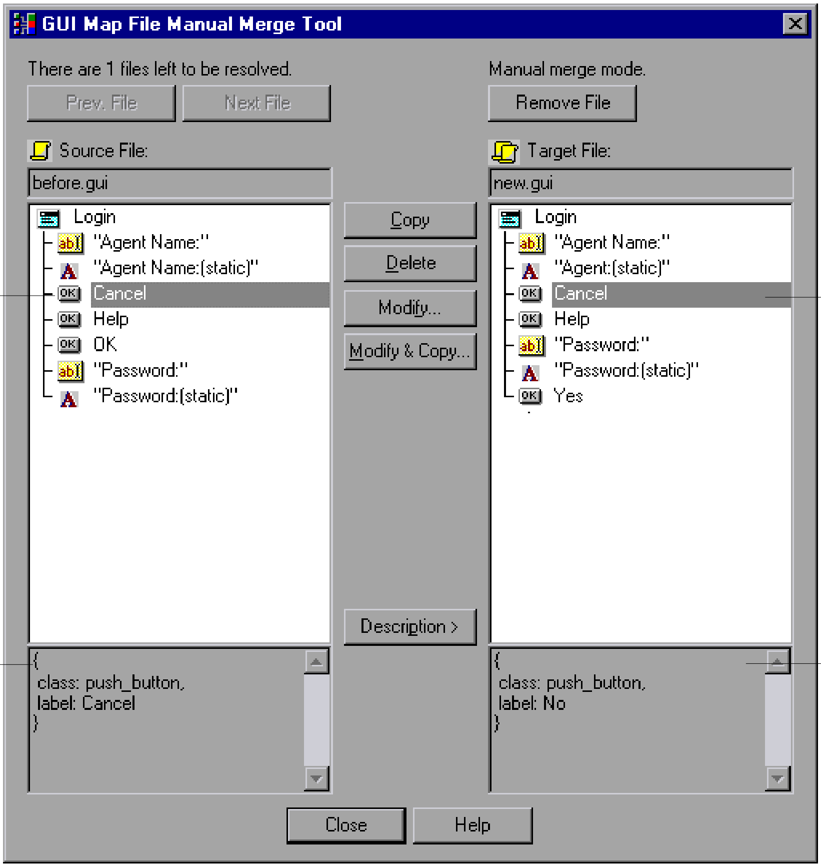
When you merge GUI map files manually, you merge each target file with the source file. The merge tool prevents you from creating conflicts while merging the files.

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

➤ Two windows with the same name but different physical descriptions.

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

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



There are 1 files left to be resolved.

Manual merge mode.

Source File: before.gui

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

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

Target File: new.gui

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

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

Logical name

Logical name

Physical description

Physical description

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

To merge GUI map files manually:

1. Follow the procedure described in "Preparing to Merge GUI Map Files" on page 98 and choose **Manual Merge** as the merge type. After you specify the source and target files and click **OK**, the GUI Map File Manual Merge Tool opens.

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

2. Locate the windows or objects to merge.

   ➤ You can double-click windows to see the objects in the window.

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

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

3. Merge the files using the following merge options:

Merge Option	Description
Copy (enabled only when an object/window in the current source file is highlighted)	Copies the highlighted object/window in source file to the highlighted window or to the parent window of the highlighted object in the target file. **Note:** Copying a window also copies all objects within that window.
Delete (enabled only when an object/window in the target file is highlighted)	Deletes the highlighted object/window from the target GUI map file. **Note:** Deleting a window also deletes all objects within that window.

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

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

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

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

Changing to the GUI Map File per Test Mode

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

In addition, you must also make the following changes:

- ➤ You should modify your tests or your startup test to load the GUI map files. For information on loading GUI map files, see “Loading the GUI Map File” on page 61.
- ➤ You must select **Global GUI Map File** in the **GUI Files** section in the **General** category of the General Options dialog box.

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

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

  For additional information on the General Options dialog box, see Chapter 41, “Setting Global Testing Options.”
- ➤ You should remember to save changes you make to GUI map files once you switch GUI map file modes. For additional information, see “Saving the GUI Map” on page 59.

9

Configuring the GUI Map

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

This chapter describes:

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

About Configuring the GUI Map

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

When WinRunner learns the description of a GUI object, it does not learn all its properties. Instead, it learns the minimum number of properties to provide a unique identification of the object.

For each object class (such as push_button, list, window, or menu), WinRunner learns a default set of properties: its GUI map configuration.

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

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

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

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

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

Note: If your application contains owner-drawn custom buttons, you can map them all to one of the standard button classes instead of mapping each button separately. You do this by either choosing a standard button class in the **Record owner-drawn buttons as** box in the Record category in the General Options dialog box or setting the *rec_owner_drawn* testing option with the **setvar** function from within a test script. For more information on the General Options dialog box, see Chapter 41, "Setting Global Testing Options." For more information on setting testing options with the **setvar** function, see Chapter 44, "Setting Testing Options from a Test Script."

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

Note about configuring non-standard Windows objects: You can use the GUI Map Configuration tool to modify how WinRunner recognizes objects with a window handle (HWND), such as standard Windows objects, ActiveX and Visual Basic controls, PowerBuilder objects, and some Web objects. For additional information on which Web objects are supported for the GUI Map Configuration tool, see Chapter 13, "Working with Web Objects." If you are working with a WinRunner add-in to test other objects, you can use the GUI map configuration functions, such as **set_record_attr**, and **set_record_method**. For additional information about these functions, refer to the *TSL Reference*. Some add-ins also have their owns tools for configuring how WinRunner recognizes objects in a specific toolkit. For additional information, refer to the *Read Me* file for your WinRunner add-in.

Understanding the Default GUI Map Configuration

For each class, WinRunner learns a set of default properties. Each default property is classified "obligatory" or "optional". (For a list of the default properties, see "All Properties" on page 125.)

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

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

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

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

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

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

The *index* selector uses numbers assigned at the time of creation of objects to identify the object in a window. Use this selector if the location of objects with the same description may change within a window. See "Configuring a Standard or Custom Class" on page 115 for more information.

Mapping a Custom Object to a Standard Class

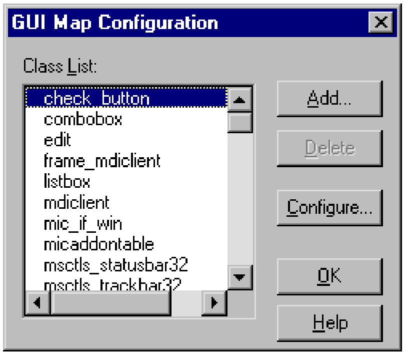
A custom object is any GUI object not belonging to one of the standard classes used by WinRunner. WinRunner learns such objects under the generic "object" class. WinRunner records operations on custom objects using **obj_mouse_** statements.

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.

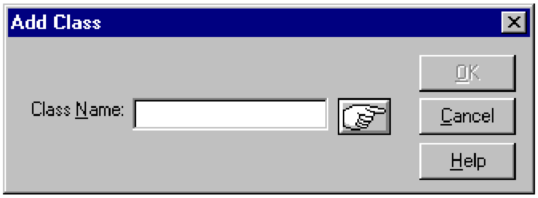
To map a custom object to a standard class:

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.

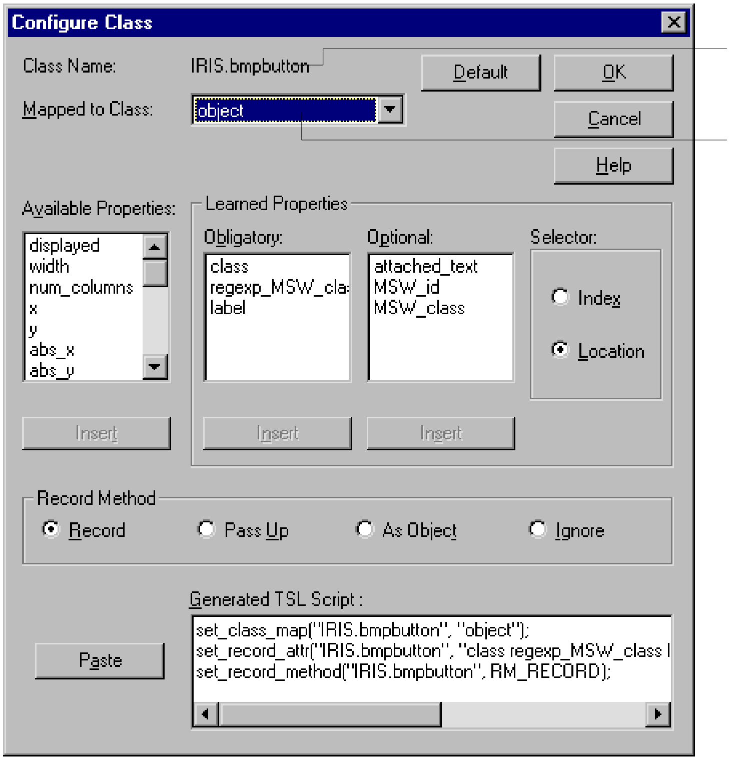
2 Click **Add** to open the Add Class dialog box.



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).

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



Class Name: IRIS.bmpbutton

Mapped to Class: object

Selector: ○ Index ● Location

Record Method: ● Record ○ Pass Up ○ As Object ○ Ignore

Generated TSL Script :

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

The custom class you are mapping

The list of standard classes

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

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

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

You can also modify the GUI map configuration of the custom class (the properties learned, the selector, or the record method). For details, see "Configuring a Standard or Custom Class" on page 115.

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 121 for more information.

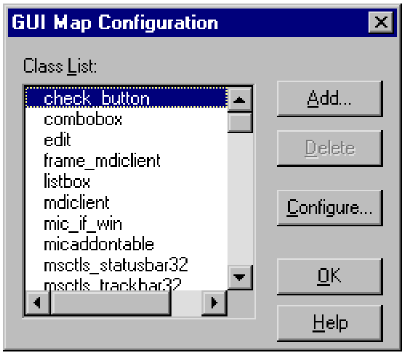
Configuring a Standard or Custom Class

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

- the properties learned
- the selector
- the recording method

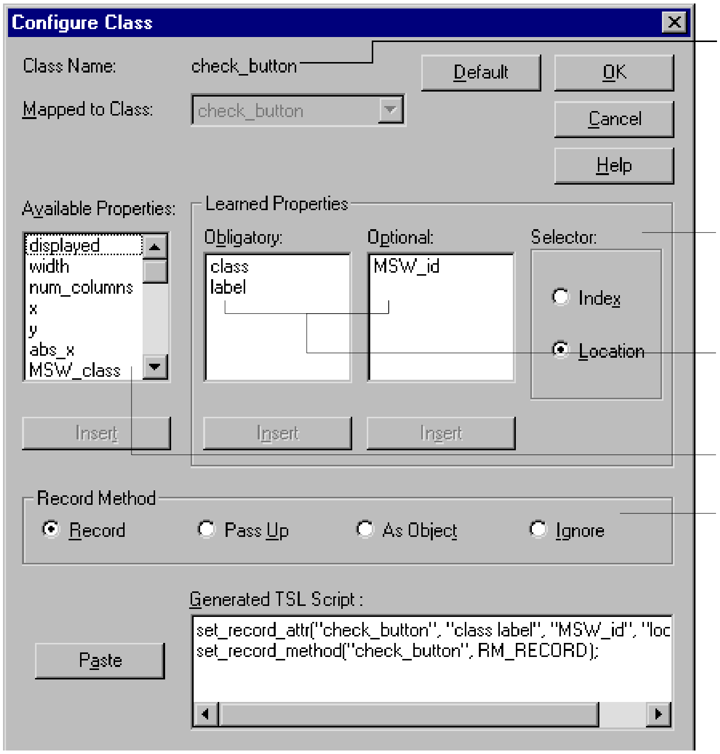
To configure a standard or custom class:

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 Name: check_button

Mapped to Class: check_button

Available Properties:
displayed
width
num_columns
x
y
abs_x
MSW_class

Learned Properties

Obligatory:
class
label

Optional:
MSW_id

Selector:
○ Index
● Location

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);
```

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 118, “Configuring the Selector” on page 120, and “Configuring the Recording Method” on page 120 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 121 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
MSW_id	All classes, except window
nchildren	All classes
obj_col_name	edit
owner	mdiclient, window
pb_name	check_button, combobox, edit, list, push_button, radio_button, scroll, window (object)
regexp_label	All classes with labels
regexp_ MSWclass	All classes
text	All classes

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

Configuring the Selector

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

A location selector performs the selection process based on the position of objects within the window: from top to bottom and from left to right. An index selector performs a selection according to a unique number assigned to an object by the application developer. For an example of how selectors are used, see "Understanding the Default GUI Map Configuration" on page 112.

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.

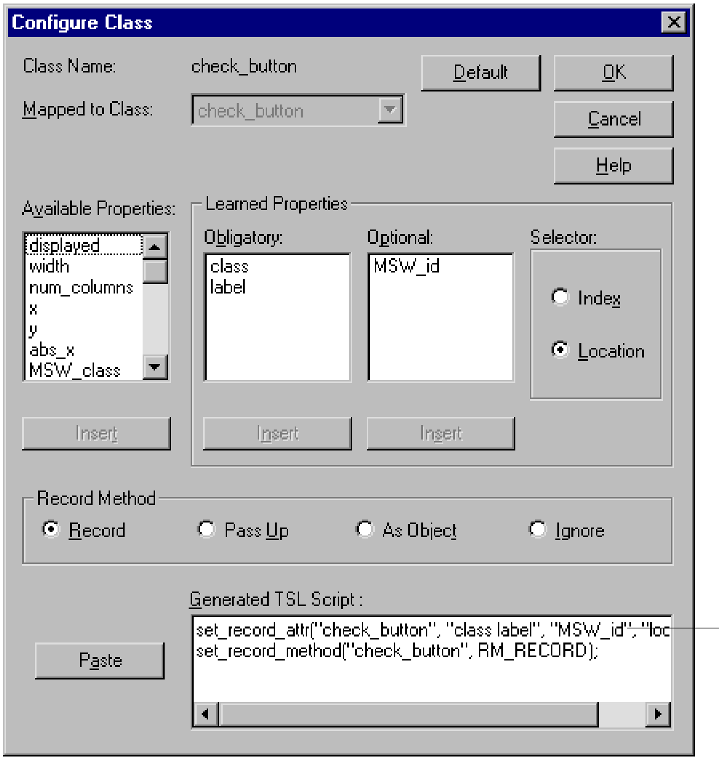
Creating a Permanent GUI Map Configuration

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.

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

Mapped to Class: check_button

Available Properties:
displayed
width
num_columns
x
y
abs_x
MSW_class

Learned Properties

Obligatory:
class
label

Optional:
MSW_id

Selector:
○ Index
● Location

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);
```

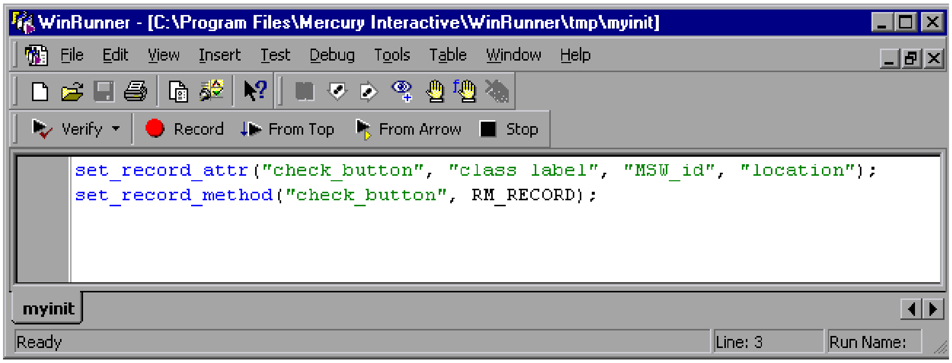
TSL statements describing the GUI map configuration

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

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

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

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



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

For more information on startup tests, see Chapter 46, “Initializing Special Configurations.” For more information on the TSL functions defining a custom GUI map configuration (**set_class_map**, **set_record_attr**, and **set_record_method**), refer to the *TSL Reference*.

Deleting a Custom Class

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

To delete a custom class:

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

The Class Property

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

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

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 map files created in XRunner in WinRunner test scripts. You must create new GUI map files in WinRunner. For information on running XRunner test scripts recorded in Analog mode, see Chapter 11, “Designing Tests.” For information on using GUI checkpoints created in XRunner in WinRunner test scripts, see Chapter 12, “Checking GUI Objects.” For information on using bitmap checkpoints created in XRunner in WinRunner test scripts, see Chapter 18, “Checking Bitmaps.”

Portable Properties

Property	Description
abs_x	The x-coordinate of the top left corner of an object, relative to the origin (upper left corner) of the screen display.
abs_y	The y-coordinate of the top left corner of an object, relative to the origin (upper left corner) of the screen display.
attached_text	The static text located near the object.
class	See “The Class Property” on page 124.
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.

Property	**Description**
focused	A Boolean value indicating whether keyboard input will be directed to this object: 1 if object has keyboard focus, 0 if not.
height	Height of object in pixels.
label	The text that appears on the object, such as a button label.
maximizable	A Boolean value indicating whether a window can be maximized: 1 if the window can be maximized, 0 if not.
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.
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.
position	The position (top to bottom) of a menu item within the menu (the first item is at position 0).
submenu	A Boolean value indicating whether a menu item has a submenu: 1 if menu has submenu, 0 if not.
value	Different for each class: Radio and check buttons: 1 if the button is checked, 0 if not. Menu items: 1 if the menu is checked, 0 if not. List objects: indicates the text string of the selected item. Edit/Static objects: indicates the text field contents. Scroll objects: indicates the scroll position. All other classes: the value property is a null string.
width	Width of object in pixels.
x	The x-coordinate of the top left corner of an object, relative to the window origin.
y	The y-coordinate of the top left corner of an object, relative to the window origin.

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

Property	Description
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.)

Default Properties Learned

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

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

Properties for Visual Basic Objects

The label and vb_name properties are obligatory properties: they are learned for all classes of Visual Basic objects. For more information on testing Visual Basic objects, see Chapter 14, "Working with ActiveX and Visual Basic Controls."

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. For more information on testing PowerBuilder objects, see Chapter 15, "Checking PowerBuilder Applications."

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*.

10

Learning Virtual Objects

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

This chapter describes:

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

About Learning Virtual Objects

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

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

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

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

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

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

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

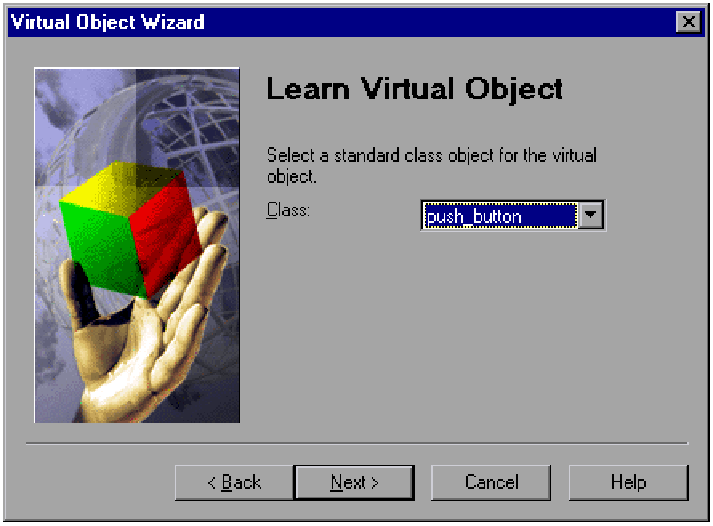
Defining a Virtual Object

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

To define a virtual object using the Virtual Object wizard:

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

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



Virtual Object Wizard

Learn Virtual Object

Select a standard class object for the virtual object.

Class: push_button

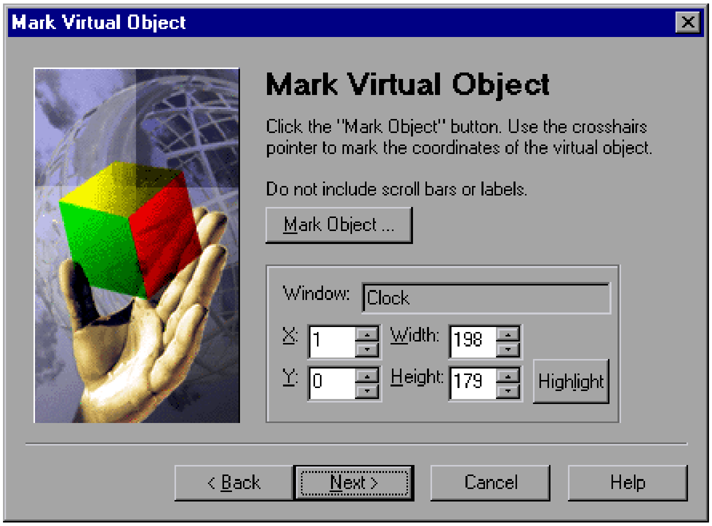
If you select the **list** class, select the number of visible rows that are displayed in the window. For a **table** class, select the number of visible rows and columns.

Click **Next**.

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

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

Press ENTER or click the right mouse button to display the virtual object's coordinates in the wizard.



Mark Virtual Object

Mark Virtual Object

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

Do not include scroll bars or labels.

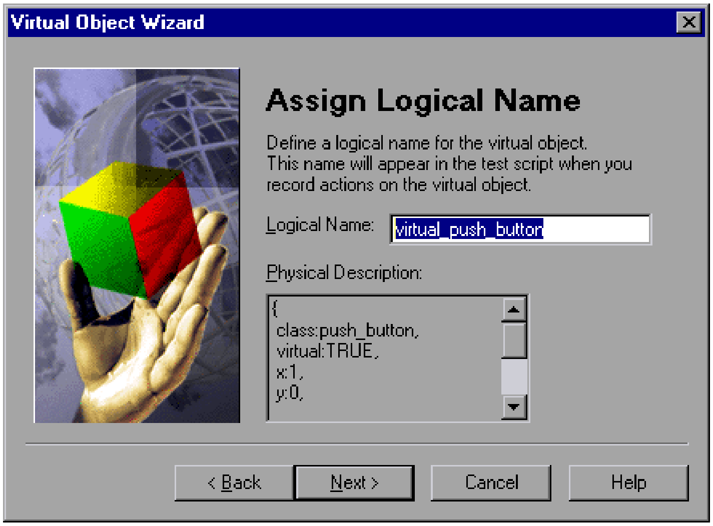
Window: Clock

X: 1 Width: 198

Y: 0 Height: 179

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

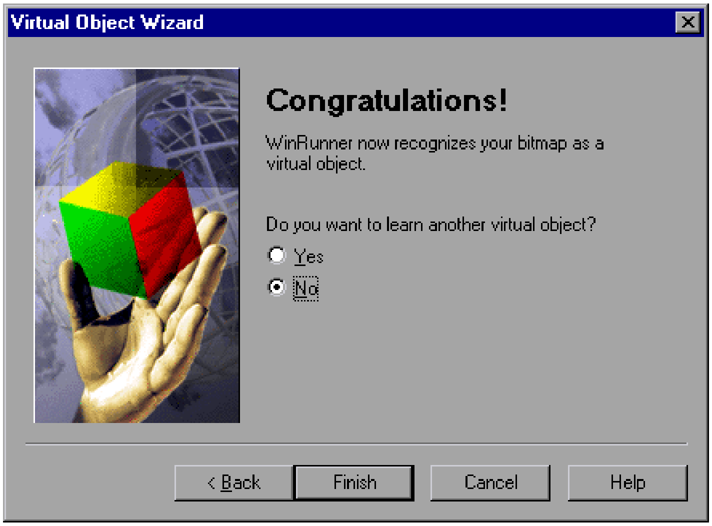
Physical Description:

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

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

5 Finish learning the virtual object:

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



Virtual Object Wizard

Congratulations!

WinRunner now recognizes your bitmap as a virtual object.

Do you want to learn another virtual object?

○ Yes

● No

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

Understanding a Virtual Object's Physical Description

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

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

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

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

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

Part III

Creating Tests

11

Designing Tests

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

This chapter describes:

- About Creating Tests
- Understanding the WinRunner Test Window
- Activating Test Creation Commands Using Softkeys
- Creating Tests Using Context Sensitive Recording
- Creating Tests Using Analog Recording
- Guidelines for Recording a Test
- Adding Checkpoints to Your Test
- Working with Data-Driven Tests
- Adding Synchronization Points to a Test
- Measuring Transactions
- Activating Test Creation Commands Using Softkeys
- Programming a Test
- Editing a Test
- Managing Test Files

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.

Note: Many WinRunner recording and editing operations are generally performed using the mouse. In accordance with Section 508, WinRunner also recognizes operations performed using the **MouseKeys** option in the Windows Accessibility Options utility. Additionally, you can perform many operations using WinRunner softkeys. For more information, see "Configuring WinRunner Softkeys," on page 850.

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

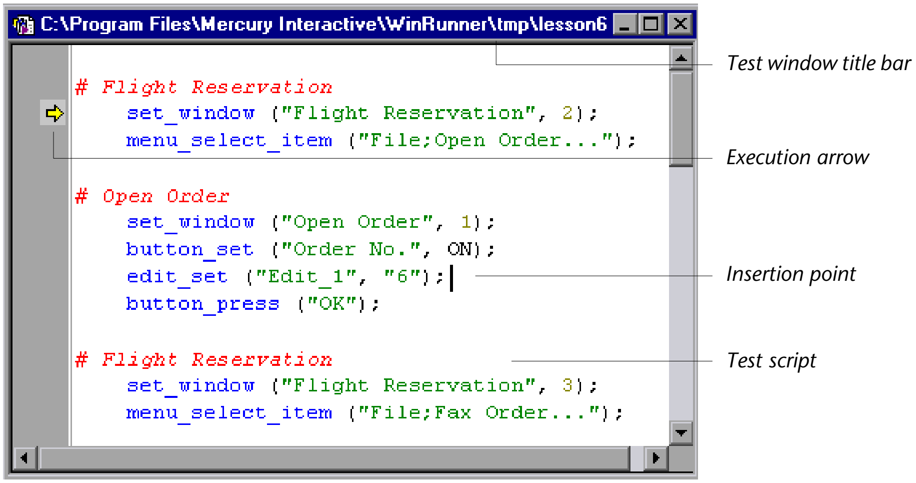
1. Decide on the functionality you want to test. Determine the checkpoints and synchronization points you need in the test script.
2. Document general information about the test in the Test Properties dialog box.
3. Choose a Record mode (*Context Sensitive* or *Analog*) and record the test on your application.
4. Assign a test name and save the test in the file system or in your TestDirector project.

Understanding 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 during a test run or the line from which the test run will begin if you use the Run test from arrow option. (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.



```
# Flight Reservation
    set_window ("Flight Reservation", 2);
    menu_select_item ("File;Open Order...");

# Open Order
    set_window ("Open Order", 1);
    button_set ("Order No.", ON);
    edit_set ("Edit_1", "6");|
    button_press ("OK");

# Flight Reservation
    set_window ("Flight Reservation", 3);
    menu_select_item ("File;Fax Order...");
```

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.
- If you plan to record some or all of your test, decide which parts of your test should use the Analog recording mode and which parts should use the Context Sensitive mode. For more information, see "Creating Tests Using Context Sensitive Recording" on page 145 and "Creating Tests Using Analog Recording" on page 150.
- Decide on the types of checkpoints and synchronization points you want to use in the test. For more information, see "Adding Checkpoints to Your Test" on page 154 and "Adding Synchronization Points to a Test" on page 155.
- Determine the types of programming elements (such as loops, arrays, and user-defined functions) that you want to add to the recorded test script. For more information, see "Programming a Test" on page 161.

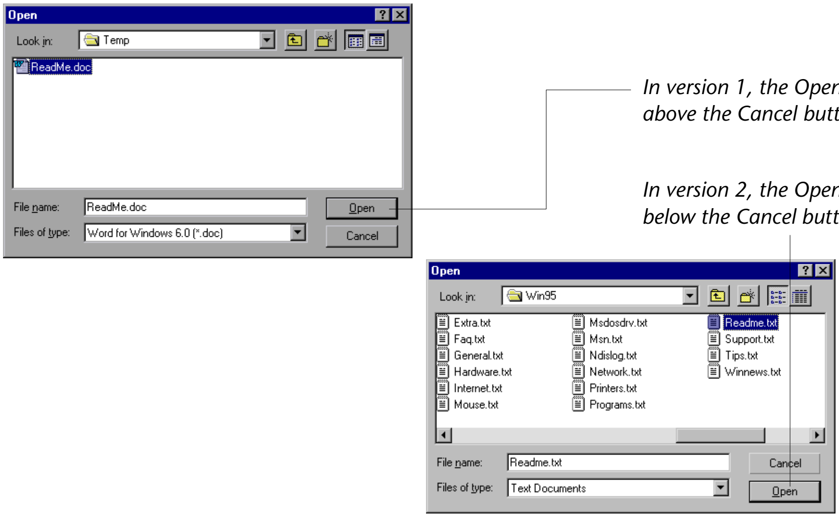
Creating Tests Using Context Sensitive Recording

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.



Look in: Temp

ReadMe.doc

File name: ReadMe.doc

Files of type: Word for Windows 6.0 (*.doc)

Look in: Win95

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

File name: Readme.txt

Files of type: Text Documents

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, and so on).

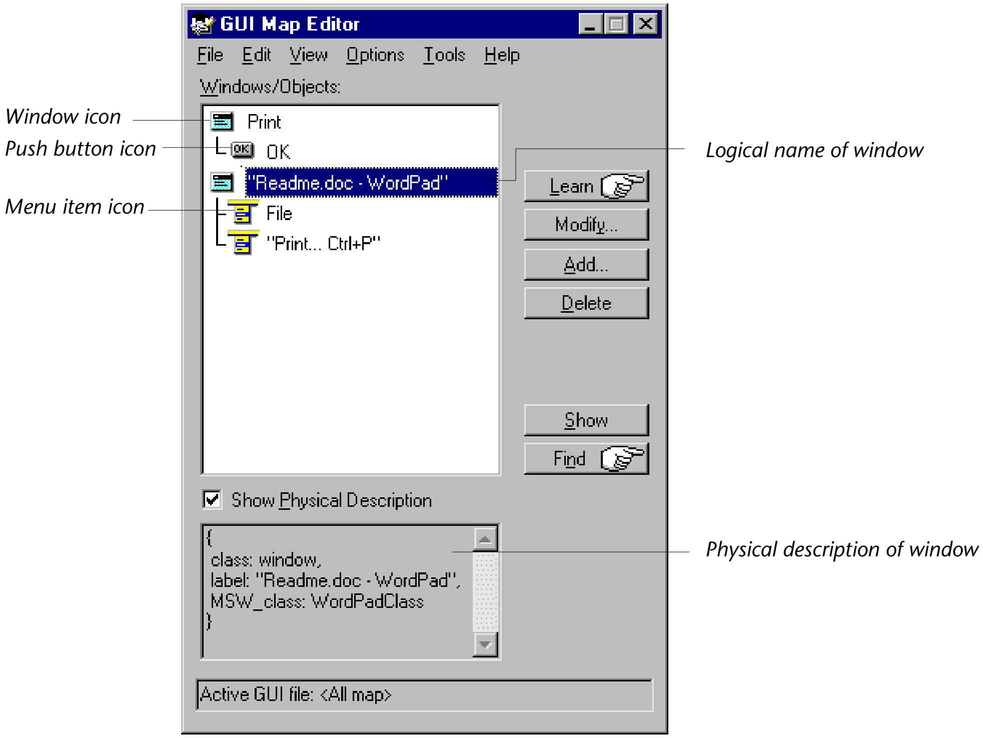
Remember that Context Sensitive tests work in conjunction with the GUI map and GUI map files. It is strongly recommended to read the "Understanding the GUI Map" section of this guide (beginning on page 23) before you start recording.

The following example illustrates the connection between the test script and the GUI map. It also demonstrates 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:

```
# Activate the Readme.doc - WordPad window.
win_activate ("Readme.doc - WordPad");

# Direct the Readme.doc - WordPad window to receive input.
set_window ("Readme.doc - WordPad", 10);

# Choose File > Print.
menu_select_item ("File;Print... Ctrl+P");

# Direct the Print window to receive input.
set_window ("Print", 10);

# Click the OK button.
button_press ("OK");
```

WinRunner learns the actual description—the list of properties and their values—for each object involved and writes this description in the GUI map.

When you open the GUI map and highlight an object, you can view the physical description. In the following example, the Readme.doc window is highlighted in the GUI map:



Window icon

Push button icon

Menu item icon

Logical name of window

Physical description of window

```
{
 class: window,
 label: "Readme.doc - WordPad",
 MSW_class: WordPadClass
}
```

WinRunner writes the following descriptions for the other window and objects in the GUI map:

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}

(To see these descriptions, you would highlight the windows or objects in the GUI map in order to see the corresponding physical description below.) 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.

To record a test in context sensitive mode:

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

The letters Rec are displayed in dark blue text with a light blue background on the **Record** button to indicate that a context sensitive record session is active.

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 **Insert** menu menu: GUI Checkpoint, Bitmap Checkpoint, Database Checkpoint, or Synchronization Point.

3 To stop recording, click **Test** > **Stop Recording** or click **Stop**.

Solving Common Context Sensitive Recording Problems

This section discusses common problems that can occur while creating Context Sensitive tests.

WinRunner Does Not Record the Appropriate TSL Statements for Your Object

You record on an object, but WinRunner does not record the appropriate TSL statements for the object class. Instead, WinRunner records **obj_mouse** statements. This occurs when WinRunner does not recognize the class to which your object belongs, and therefore it assigns it to the generic "object" class.

There are several possible causes and solutions:

Possible Causes	**Possible Solutions**
Add-in support for the object is not loaded.	You must install and load add-in support for the required object. For example, for HTML objects, you must load the WebTest add-in. For information on loading add-in support, see "Loading WinRunner Add-Ins" on page 20.
The object is a custom class object.	If a custom object is similar to a standard object, you can map the custom class to a standard class, as described in "Mapping a Custom Object to a Standard Class" on page 113.
	You can add a custom GUI object class. For more information on creating custom GUI object classes and checking custom objects, refer to the *WinRunner Customization Guide*. You can also create GUI checks for custom objects. For information on checking GUI objects, see Chapter 5, "Working in the Global GUI Map File Mode."
	You can create custom record and execution functions. If your object changes, you can modify your functions instead of updating all your test scripts. For more information on creating custom record and execution functions, refer to the *WinRunner Customization Guide*.

WinRunner Cannot Read Text from HTML Pages in Your Application

There are several possible causes and solutions:

Possible Causes	**Possible Solutions**
The WebTest add-in is not loaded.	You must install and load add-in support for Web objects. For information on loading add-in support, see "Loading WinRunner Add-Ins" on page 20.

Possible Causes	**Possible Solutions**
WinRunner does not identify the text as originating in an HTML frame or table.	Use the **Insert** > **Get Text** > **From Selection (Web only)** command to retrieve text from an HTML page. For a frame, WinRunner inserts a **web_frame_get_text** statement. For any other GUI object class, WinRunner inserts a **web_obj_get_text** statement.
	Use the **Insert** > **Get Text** > **Web Text Checkpoint** command to check whether a specified text string exists in an HTML page. For a frame, WinRunner inserts a **web_frame_text_exists** statement. For any other GUI object class, WinRunner inserts a **web_obj_text_exists** statement.

For more information, see Chapter 13, "Working with Web Objects," or the *TSL Reference*.

For more information on solving Context Sensitive testing problems, refer to WinRunner context-sensitive help.

Creating Tests Using 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>+");
```

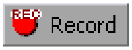
Use Analog mode when exact mouse movements are an integral part of the test, such as in a drawing application. Note that you can switch to and from Analog mode during a Context Sensitive recording session by selecting the appropriate many item, clicking the **Record** button during the record session, or using the F2 shortcut key.

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 9, “Configuring the GUI Map.” For information on using GUI checkpoints created in XRunner in WinRunner test scripts, see Chapter 12, “Checking GUI Objects.” For information on using bitmap checkpoints created in XRunner in WinRunner test scripts, see Chapter 18, “Checking Bitmaps.”

To record a test using Analog mode:

1 Position the WinRunner window and the application you are testing so that you can see both applications.

2 Choose **Test** > **Record – Analog.** Alternatively, click the **Record–Context Sensitive** button to start recording in context sensitive mode, and then click the **Record** button again or press F2 any time during the recording session to toggle to analog mode.



The letters Rec are displayed in red text with a white background on the **Record** button to indicate that an analog record session is active.

3 Perform the necessary operations on the application you want to test using the keyboard and mouse.

Note: All mouse operations, including those performed on the WinRunner window or WinRunner dialog boxes are recorded during an analog recording session. Therefore, you should not insert checkpoints and synchronization points, or select other WinRunner menu or toolbar options during an analog recording session.

---

4 To stop recording, click **Test** > **Stop Recording** or click **Stop**. To switch back to context-sensitive recording mode, press F2 or click the **Record** toolbar button.

Guidelines for Recording a Test

Consider the following guidelines when recording a test:

- Before you start to record, close all applications not required for the test.
- Use an **invoke_application** statement or set a startup application in the Run tab of the Test Properties dialog box to open the application you are testing.

  For information on working with TSL functions, see Chapter 26, "Enhancing Your Test Scripts with Programming." For more information about the **invoke_application** function and an example of usage, refer to the *TSL Reference*. For more information on startup applications, see "Defining Startup Applications and Functions" on page 758.
- Before you record on objects within a window, click the title bar of the window to record a **win_activate** statement. This activates the window. For information on working with TSL functions, see Chapter 26, "Enhancing Your Test Scripts with Programming." For more information about the **win_activate** function and an example of usage, refer to the *TSL Reference*.
- 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 43, "Customizing the WinRunner 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 41, "Setting Global Testing Options," and Chapter 44, "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 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.

Adding Checkpoints to Your Test

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 12, "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 18, "Checking Bitmaps," for more information.
- Text checkpoints read text in GUI objects and in bitmaps and enable you to verify their contents. See Chapter 19, "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 17, "Checking Databases," for more information.

Working with 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 21, "Creating Data-Driven Tests."

Adding Synchronization Points to a Test

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 22, "Synchronizing the Test Run," for more information.

Measuring Transactions

You can measure how long it takes to run a section of your test by defining transactions. A transaction represents the business process that you are interested in measuring. You define transactions within your test by enclosing the appropriate sections of the test with **start_transaction** and **end_transaction** statements. For example, you can define a transaction that measures how long it takes to reserve a seat on a flight and for the confirmation to be displayed on the client's terminal.

You must declare each transaction using a **declare_transaction** statement somewhere in the test prior to the corresponding **start_transaction** statement. You may want to declare all transactions at the beginning of your test, or you can declare each transaction immediately prior to the corresponding **start_transaction** statement.

During the test run, the **start_transaction** statement signals the beginning of the time measurement. The time measurement continues until the **end_transaction** statement is encountered. The test report displays the time it took to perform the transaction.

Consider the following when planning transactions:

- There is no limit to the number of transactions that can be added to a test.
- It is recommended to insert a synchronization point before the end of the transaction.
- Transactions can be nested, but each **start_transaction** statement must be associated with a corresponding **end_transaction** statement.

Notes:

If no **end_transaction** statement exists for a particular transaction, then no transaction time is reported to the test results.

If a **start_transaction** name is used more than once before the corresponding **end_transaction**, then the timing restarts (reset to 0) when the test run reaches the line containing the repeated **start_transaction** statement.

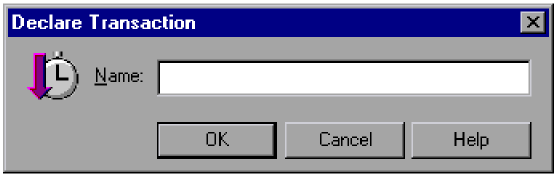
You can insert **declare_transaction**, **start_transaction**, and **end_transaction** statements manually, or you can use the **Insert > Transactions** options to insert these statements.

To insert transaction statements using the Insert > Transactions options:

1 If you want to insert the **declare_transaction** and **start_transaction** statements on consecutive lines, proceed to step 4.

If you want to insert the **declare_transaction** statement two or more lines above the **start_transaction** statement, place the cursor at the location where you want to declare the transaction.

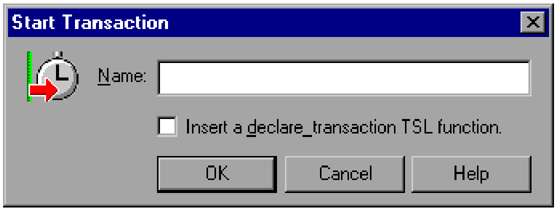
2 Choose **Insert** > **Transactions** > **Declare Transaction**. The Declare Transaction dialog box opens.



3 Enter a name for the transaction and click OK. The **declare_transaction** statement is added to your test.

4 Place the cursor at the beginning of the line where you want the transaction measurement to begin.

5 Choose **Insert** > **Transactions** > **Start Transaction**. The Start Transaction dialog box opens.



6 Enter a name for the transaction.

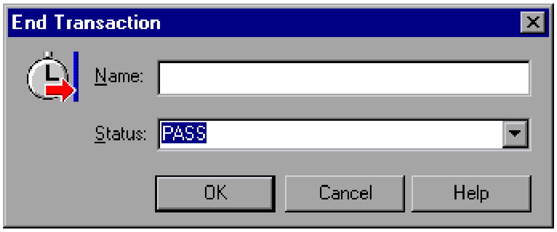
If you have already entered a **declare_transaction** statement in the test, the **start_transaction** name should be identical to the one specified in the **declare_transaction** statement. Note that transaction names are case-sensitive.

7 If you have not yet entered a **declare_transaction** statement for this transaction, and you want to insert the declaration on the line immediately above the **start_transaction** statement, select the **Insert a declare_transaction TSL function** check box.

8 Click **OK**. The **start_transaction** (and **declare_transaction**, if applicable) statement(s) are added to your test.

9 Place the cursor below the line that marks the end of the transaction measurement.

10 Choose **Insert** > **Transactions** > **End Transaction**. The End Transaction dialog box opens.



11 Enter the name of the transaction you want to end. The transaction name must be identical to the name used in the **declare_transaction** and **start_transaction** statements. Note that transaction names are case-sensitive.

12 Select the pass/fail status that you want to assign to the transaction.

13 Click **OK**.

For information on inserting **declare_transaction**, **start_transaction**, and **end_transaction** statements manually, refer to the *TSL Reference*.

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 43, "Customizing the WinRunner 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.
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.
RUNTIME RECORD CHECK	Alt Right + F10	Opens the Runtime Wizard Checkpoint Wizard.
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.

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.
CALL QUICKTEST TEST	Ctrl Left + q	Inserts a call to a QuickTest test.
DECLARE TRANSACTION	Ctrl Left + 4	Inserts a **declare_transaction** statement.
START TRANSACTION	Ctrl Left + 5	Inserts a **start_transaction** statement.
END TRANSACTION	Ctrl Left + 6	Inserts an **end_transaction** statement.
DATA TABLE	Ctrl Left + 8	Opens an existing data table or creates a new one.
PARAMETERIZE DATA	Ctrl Left + 9	Opens the Parameterize Data dialog box.
DATA DRIVER WIZARD	Ctrl Left + 0	Opens the Data Driver Wizard.
STOP	Ctrl Left + F3	Stops test recording.

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 27, “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.

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.
Redo	Reverses the last Undo 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.
Comment	Converts the selected line(s) of text to a comment by adding a ‘#’ sign at the beginning of the line. The commented text is also converted to italicized, red text.

Edit Command	Description
Uncomment	Converts the selected, commented line(s) of text into executable code by removing the '#' sign from the beginning of the line. The text is also converted to plain, black text.
Increase Indent	Moves the selected line(s) of text one tab stop to the right. Note that you can change the tab stop size in the Editor Options dialog box. For more information, see page 822.
Decrease Indent	Moves the selected line(s) of text one tab stop to the left. Note that you can change the tab stop size in the Editor Options dialog box. For more information, see page 822.
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.

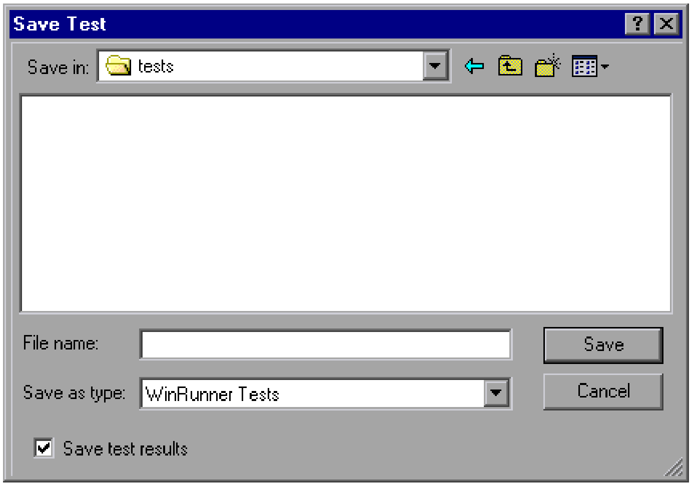
Saving a Test

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**.

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: tests

File name:

Save as type: WinRunner Tests

☑ Save test results

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. Select or clear the **Save test results** check box to indicate whether you want to save any existing test results with your test.

   Note that if you clear this box, your test result files will not be saved with the test, and you will not be able to view them later. Clearing the **Save test results** check box can be useful for conserving disk space if you do not require the test results for later analysis, or if you are saving an existing test under a new name and do not need the test results.

   Note: By default, this option is selected when saving a new test (**Save**), and cleared when saving an existing test under a new name (**Save As**).

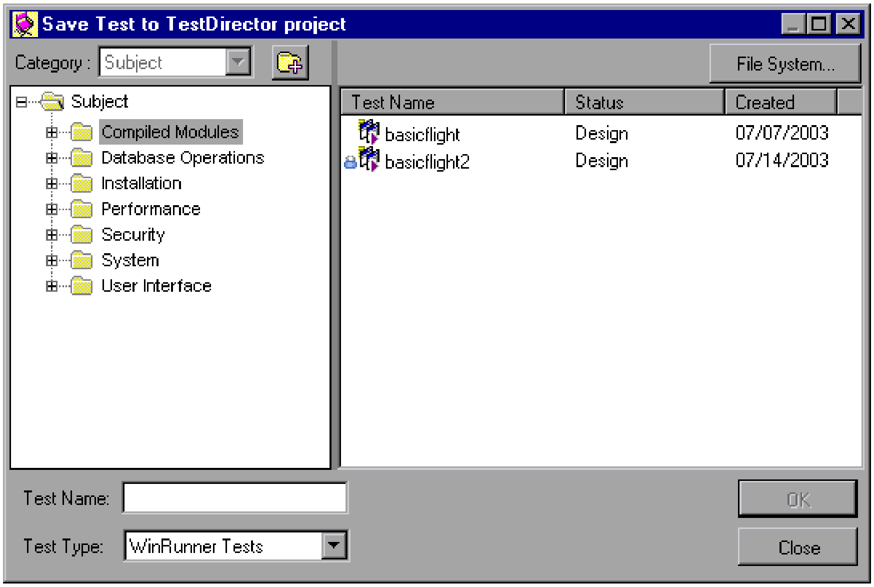
5. Click **Save** to save the test.

To save a test to a TestDirector project:

Note: You can only save a test to a TestDirector database if you are working with TestDirector. For additional information, see Chapter 48, "Managing the Testing Process."

---

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.



Test Name	Status	Created
basicflight	Design	07/07/2003
basicflight2	Design	07/14/2003

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 or click the **New Folder** button to create a new subject folder. To expand the subject tree, double-click a closed folder icon. To collapse a sublevel, double-click an open folder icon.

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 48, "Managing the Testing Process."

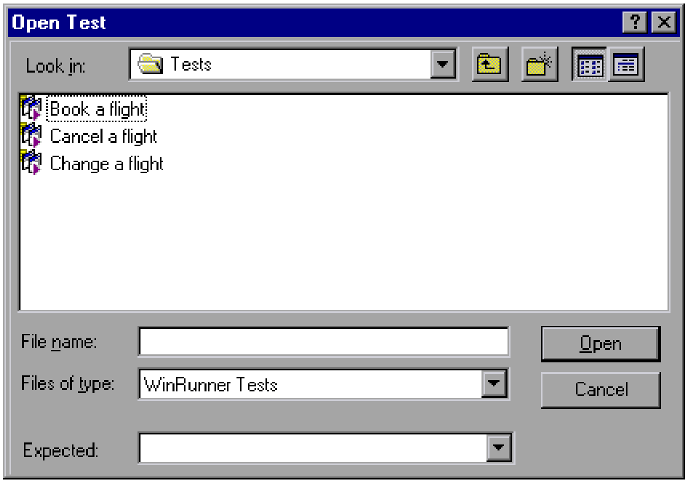
Opening an Existing Test

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.



Open Test

Look in: Tests

Book a flight
Cancel a flight
Change a flight

File name:

Files of type: WinRunner Tests

Expected:

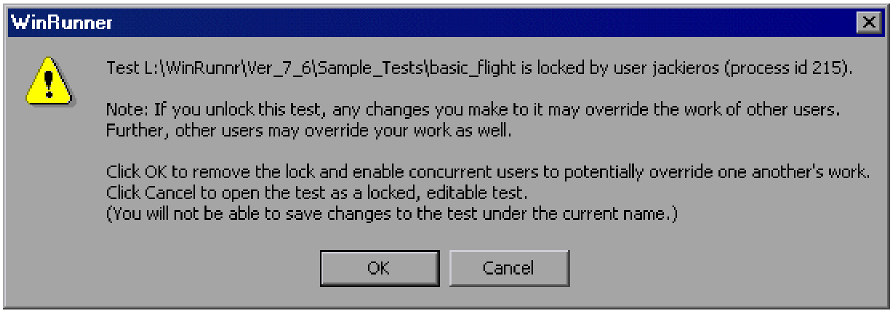
2 In the **Look in** box, click the location of the test you want to open.

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.

If you select to open a test that is already opened by another WinRunner user, a message similar to the following opens:



Test L:\WinRunnr\Ver_7_6\Sample_Tests\basic_flight is locked by user jackieros (process id 215).

Note: If you unlock this test, any changes you make to it may override the work of other users. Further, other users may override your work as well.

Click OK to remove the lock and enable concurrent users to potentially override one another's work.
Click Cancel to open the test as a locked, editable test.
(You will not be able to save changes to the test under the current name.)

Click **Cancel** to open the test as a locked, editable test. You can edit and run the test, but you cannot save the test with its current name.

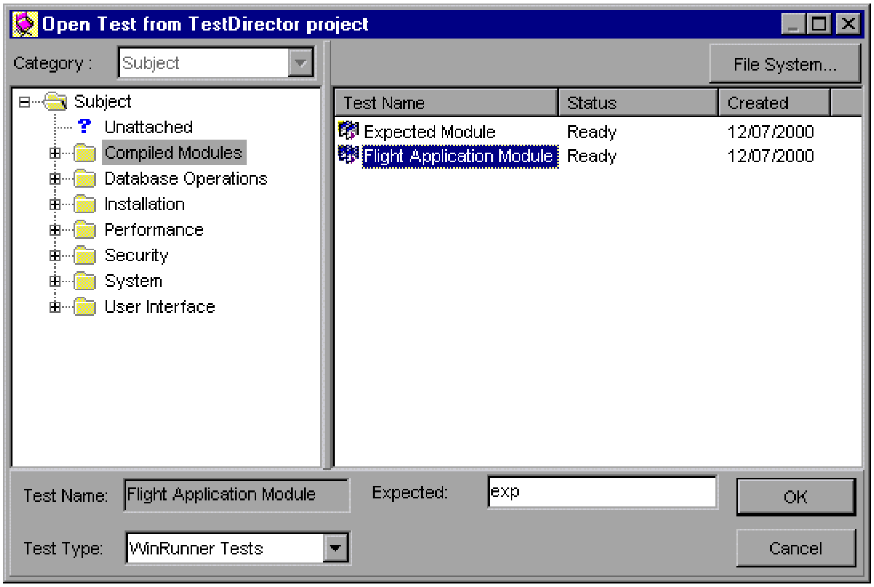
Click **OK** to unlock the test only if you are sure that your work will not interfere with other users.

To open a test from a TestDirector project:

Note: You can only open a test from a TestDirector database if you are working with TestDirector. For additional information, see Chapter 48, "Managing the Testing Process."

---

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.



Test Name	Status	Created
Expected Module	Ready	12/07/2000
Flight Application Module	Ready	12/07/2000

Test Name: Flight Application Module

Expected: exp

Test Type: WinRunner Tests

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 48, "Managing the Testing Process."

Zipping and Extracting WinRunner Tests

You can zip your WinRunner test for easy distribution using the **Export to Zip File** option. When you choose this option, all files that are saved in your test folder are zipped, including the Data Table, test results, GUI files, etc. External files stored in locations outside your test folder are not zipped.

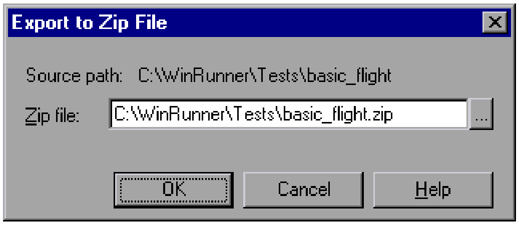
You can use the **Import from Zip File** option to extract the files for any test that was zipped using the **Export to Zip File** option. Note that you cannot use this option to extract files from a test that was zipped using another utility.

To zip a test:

1 Open the test you want to zip.

2 If the open test contains unsaved changes, save the test.

3 Choose **File** > **Export to Zip File**. The Export to Zip File dialog box opens and displays the source path of the test and a suggested zip file name.


Export to Zip File

Source path: C:\WinRunner\Tests\basic_flight

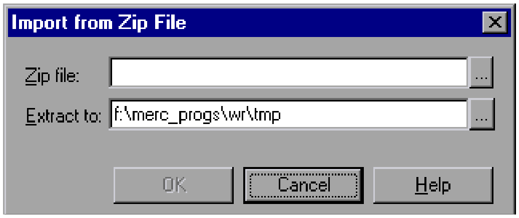
Zip file: C:\WinRunner\Tests\basic_flight.zip

4 Accept the default zip file name and path or specify a new one.

5 Click **OK**. The dialog box displays a progress bar as it zips the test. The dialog box closes when the zip process is complete.

To extract a zipped test:

1 Choose **File** > **Import from Zip File**. The Import from Zip File dialog box opens.


Import from Zip File

Zip file:

Extract to: f:\merc_progs\wr\tmp

2 Enter or browse to the location of the zipped test you want to extract.

3 Accept the default location for extracting the test, or specify a new location.

4 Click **OK**. The dialog box displays a progress bar as it extracts the test. When the extraction process is complete, the dialog box closes and the extracted test is displayed in the WinRunner window.

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 **File** > **Close All**.

12

Checking GUI Objects

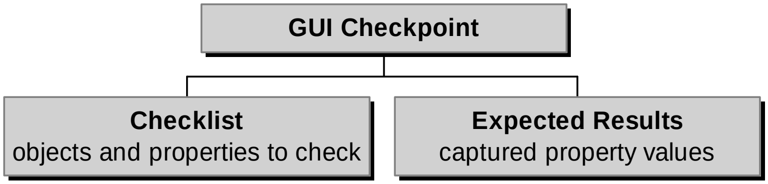
By adding GUI checkpoints to your test scripts, you can compare the behavior of GUI objects in different versions of your application.

This chapter describes:

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.



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. Your GUI checkpoint can be part of a loop. If a GUI checkpoint is run in a loop, the results for each iteration of the checkpoint are displayed in the test results as separate entries. The results of each iteration of the checkpoint can be viewed in the Test Results window. For more information, see Chapter 34, "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, "Understanding How WinRunner Identifies GUI Objects," 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 25, "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 14, "Working with ActiveX and Visual Basic Controls." For information on WinRunner support for PowerBuilder, see Chapter 15, "Checking PowerBuilder Applications."

You can also create GUI checkpoints that check the contents and properties of tables. For information, see Chapter 16, "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 9, "Configuring the GUI Map." For information on using test scripts recorded in XRunner in Analog mode, see Chapter 11, "Designing Tests." For information on using bitmap checkpoints created in XRunner, see Chapter 18, "Checking Bitmaps."

Setting Options for Failed GUI Checkpoints

You can instruct WinRunner to send an e-mail to selected recipients each time a GUI checkpoint fails and you can instruct WinRunner to capture a bitmap of your window or screen when any checkpoint fails. You set these options in the General Options dialog box.

To instruct WinRunner to send an e-mail message when a GUI checkpoint fails:

1 Choose **Tools** > **General Options**. The General Options dialog box opens.

2 Click the **Notifications** category in the options pane. The notification options are displayed.

3 Select **GUI checkpoint failure**.

4 Click the **Notifications** > **E-mail** category in the options pane. The e-mail options are displayed.

5 Select the **Active E-mail service** option and set the relevant server and sender information.

6 Click the **Notifications** > **Recipient** category in the options pane. The e-mail recipient options are displayed.

7 Add, remove, or modify recipient details as necessary to set the recipients to whom you want to send an e-mail message when a GUI checkpoint fails.

The e-mail contains summary details about the test and checkpoint and details about the expected and actual values of the property check.

For more information, see "Setting Notification Options" on page 808.

To instruct WinRunner to capture a bitmap when a checkpoint fails:

1 Choose **Tools** > **General Options**. The General Options dialog box opens.

2 Click the **Run** > **Settings** category in the options pane. The run settings options are displayed.

3 Select **Capture bitmap on verification failure**.

4 Select **Window**, **Desktop**, or **Desktop area** to indicate what you want to capture when checkpoints fail.

5 If you select **Desktop area**, specify the coordinates of the area of the desktop that you want to capture.

When you run your test, the captured bitmaps are saved in your test results folder.

For more information, see "Setting Test Run Options" on page 793.

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

edit_check_info

list_check_info

obj_check_info

scroll_check_info

static_check_info

win_check_info

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

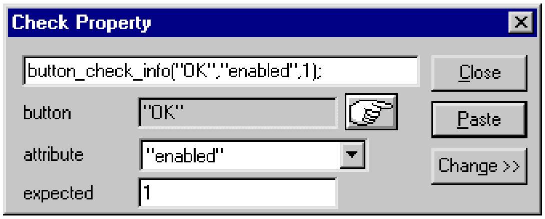
To create a GUI checkpoint for a property value:

1. Choose **Insert** > **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.

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.



button_check_info("OK","enabled",1);

button "OK"

attribute "enabled"

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 27, "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 207.

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 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 **Insert** > **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 187.

Creating a GUI Checkpoint by Specifying which Properties to Check

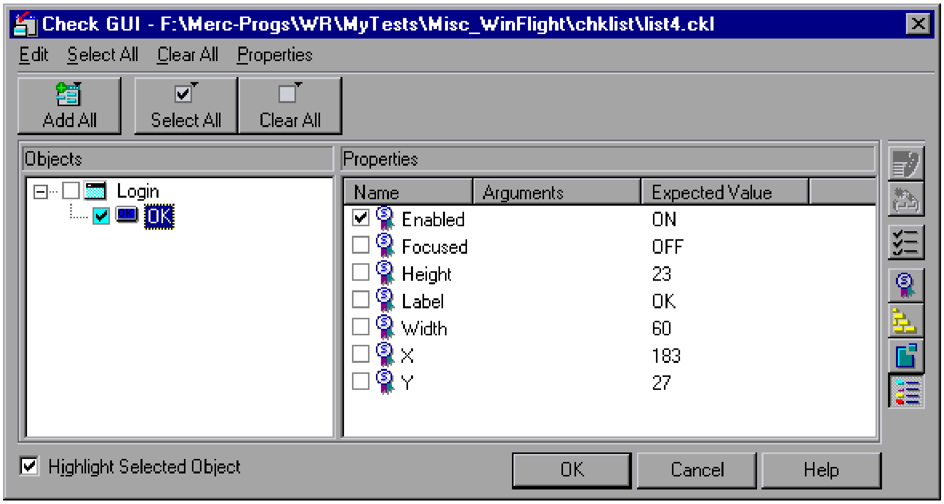
You can specify which properties to check for an object. For example, if you create a checkpoint that checks a push button, you can choose to verify that it is in focus, instead of enabled.

To create a GUI checkpoint by specifying which properties to check:

1 Choose **Insert** > **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 Double-click the object or window. The Check GUI dialog box opens.


Check GUI - F:\Merc-Progs\WR\MyTests\Misc_WinFlight\chklist\list4.ckl
Edit
Select All
Clear All
Properties
Add All
Select All
Clear All
Objects
Login
OK
Properties
☑ Highlight Selected Object
OK
Cancel
Help

Name	Arguments	Expected Value
☑ Enabled		ON
□ Focused		OFF
□ Height		23
□ Label		OK
□ Width		60
□ X		183
□ Y		27

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 219.
- 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 an ellipsis (three dots) appears 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 213.
- 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 198.

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 187.

For more information on the Check GUI dialog box, see “Understanding the GUI Checkpoint Dialog Boxes” on page 196.

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 207.

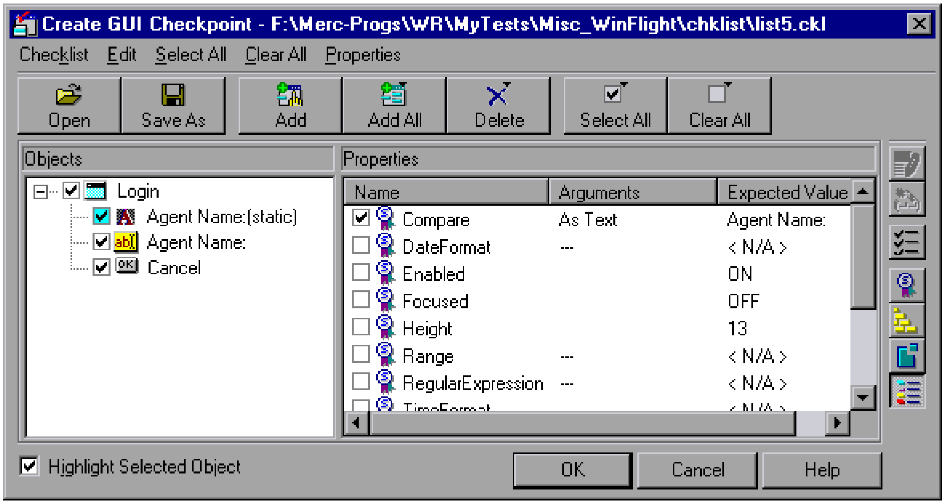
To create a GUI checkpoint for two or more objects:

1. Choose **Insert** > **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 184.
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.



Create GUI Checkpoint - F:\Merc-Progs\WR\MyTests\Misc_WinFlight\chklist\list5.ckl

Objects:

- ☑ Login
  - ☑ Agent Name:(static)
  - ☑ Agent Name:
  - ☑ Cancel

	Name	Arguments	Expected Value
☑	Compare	As Text	Agent Name:
□	DateFormat	---	< N/A >
□	Enabled		ON
□	Focused		OFF
□	Height		13
□	Range	---	< N/A >
□	RegularExpression	---	< N/A >
□	TimeFormat		< N/A >

☑ Highlight Selected Object

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 219.

➤ 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 an ellipsis appears 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 213.

➤ 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 201.

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 187.

For more information on the Create GUI Checkpoint dialog box, see “Understanding the GUI Checkpoint Dialog Boxes” on page 196.

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 207.

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 Reference* and the *WinRunner Customization Guide*.

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.

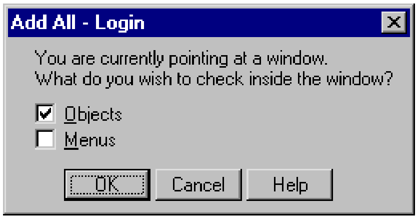
To create a GUI checkpoint that performs a default check on every GUI object in a window:

1. Choose **Insert** > **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.



Add All - Login

You are currently pointing at a window.
What do you wish to check inside the window?

☑ Objects
□ Menus

3. Select **Objects** or **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 **Insert** > **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.

3. Specify which checks to perform, and click **OK** to close the dialog box. For more information, see "The Check GUI Dialog Box" on page 198.

   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 188.
- ➤ 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 630. 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* **);**

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 44, "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 44, "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 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 61.

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 190.

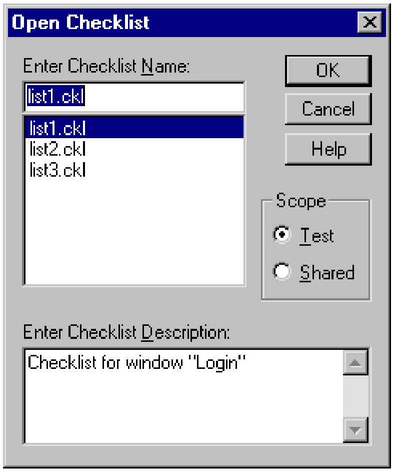
To use an existing GUI checklist in a GUI checkpoint:

1 Choose **Insert** > **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

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 221.

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** category of the General Options dialog box. For more information, see Chapter 41, "Setting Global Testing Options."

To save a GUI checklist in a shared folder:

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

The Open Checklist dialog box opens. Note that GUI checklists have the .ckl extension, while database checklists have the .cdl extension.

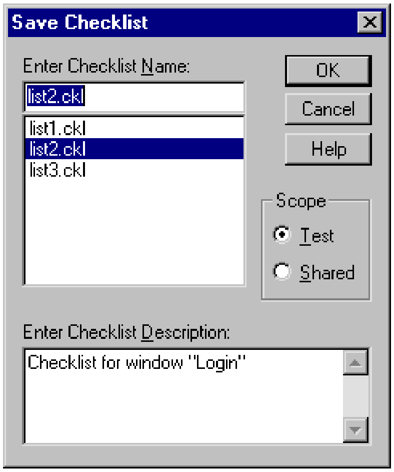
For information on database checklists, see "Modifying a Standard Database Checkpoint," on page 350.

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

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

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

The Save Checklist dialog box opens.



Save Checklist

Enter Checklist Name: list2.ckl

list1.ckl
list2.ckl
list3.ckl

Scope: ● Test ○ Shared

Enter Checklist Description: Checklist for window "Login"

4 Under **Scope**, click **Shared**.

Type 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:

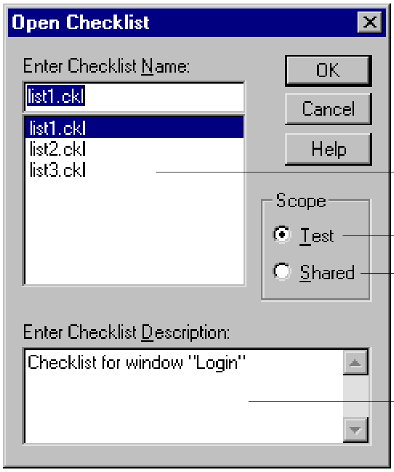
- change which objects in a window to check
- change which properties of an object to check
- change the arguments for an existing property check
- specify the arguments for a new property check

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 61.

To edit an existing GUI checklist:

1. Choose **Insert** > **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 190.



Enter Checklist Name: list1.ckl

list1.ckl
list2.ckl
list3.ckl

Scope
● Test
○ Shared

Enter Checklist Description: Checklist for window "Login"

Lists the available checklists.

Displays checklists created for the current test.

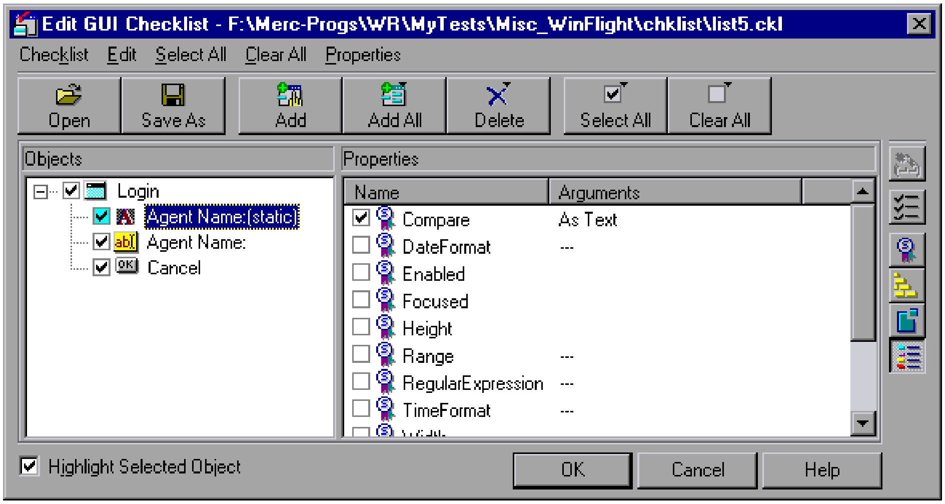
Displays checklists created in a shared folder.

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.



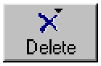
☑ Login
- ☑ Agent Name:(static)
- ☑ Agent Name:
- ☑ Cancel

Name	Arguments
☑ Compare	As Text
□ DateFormat	---
□ Enabled	
□ Focused	
□ Height	
□ Range	---
□ RegularExpression	---
□ TimeFormat	---

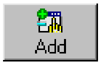
☑ Highlight Selected Object

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 204.

- 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.

- 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 an ellipsis appears 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 213.

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.

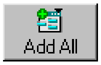
For more information on the Edit GUI Checklist dialog box, see “Understanding the GUI Checkpoint Dialog Boxes” on page 196.

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 621.

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

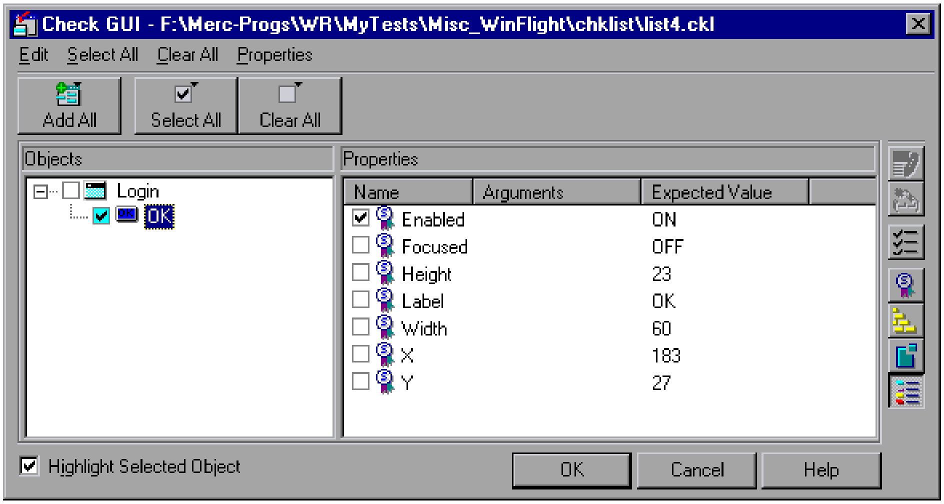
The following messages may appear in the GUI Checkpoint dialog boxes:

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
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

Note: For information on the GUI Checkpoint Results dialog box, see "Modifying the Expected Results of a GUI Checkpoint" on page 221 or Chapter 34, "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 **Insert** > **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.



Objects:
- □ Login
  - ☑ OK

Name	Arguments	Expected Value
☑ Enabled		ON
□ Focused		OFF
□ Height		23
□ Label		OK
□ Width		60
□ X		183
□ Y		27

☑ Highlight Selected Object

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 checkmark 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:

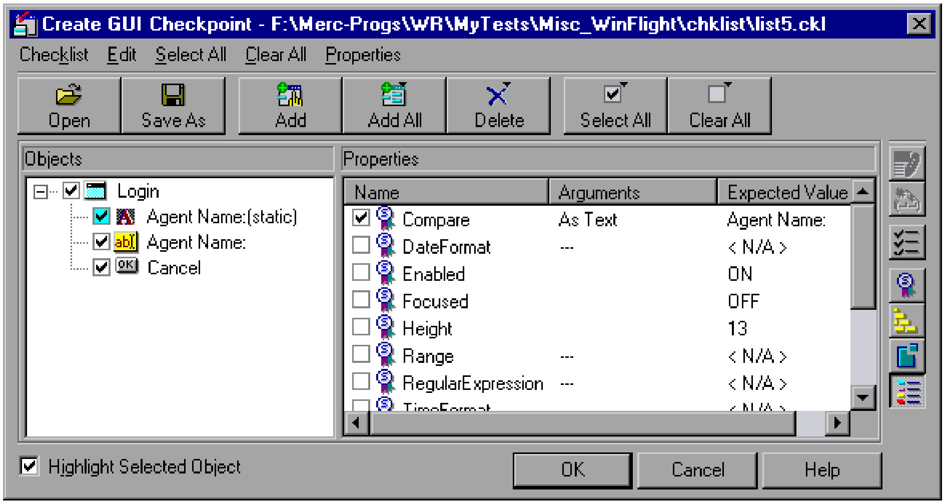
Button	Description
Add All	**Add All** adds all objects or menus in a window to your checklist.
☑ Select All	**Select All** selects all objects, properties, or objects of a given class in the Check GUI dialog box. If you want to select all objects of a given class, the Classes of Objects dialog box opens. 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*.
EXP	**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 219.
	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 213.

Button	Description
☑	**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.
S	**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 **Insert** > **GUI Checkpoint** > **For Multiple Objects** or click the **GUI Checkpoint for Multiple Objects** button on the User toolbar.



Objects:

- ☑ Login
  - ☑ Agent Name:(static)
  - ☑ Agent Name:
  - ☑ Cancel

Name	Arguments	Expected Value
☑ Compare	As Text	Agent Name:
□ DateFormat	---	< N/A >
□ Enabled		ON
□ Focused		OFF
□ Height		13
□ Range	---	< N/A >
□ RegularExpression	---	< N/A >
□ TimeFormat		< N/A >

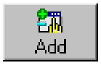
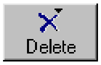
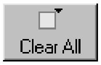
☑ Highlight Selected Object

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 checkmark 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.

The Create GUI Checkpoint dialog box includes the following options:

Button	Description
	Open opens an existing GUI checklist.
	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** adds an object to your GUI checklist.
	Add All adds all objects or menus in a window to your GUI checklist.
	**Delete** deletes an object, or all of the objects that appear in the GUI checklist.
	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.
	**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 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*.

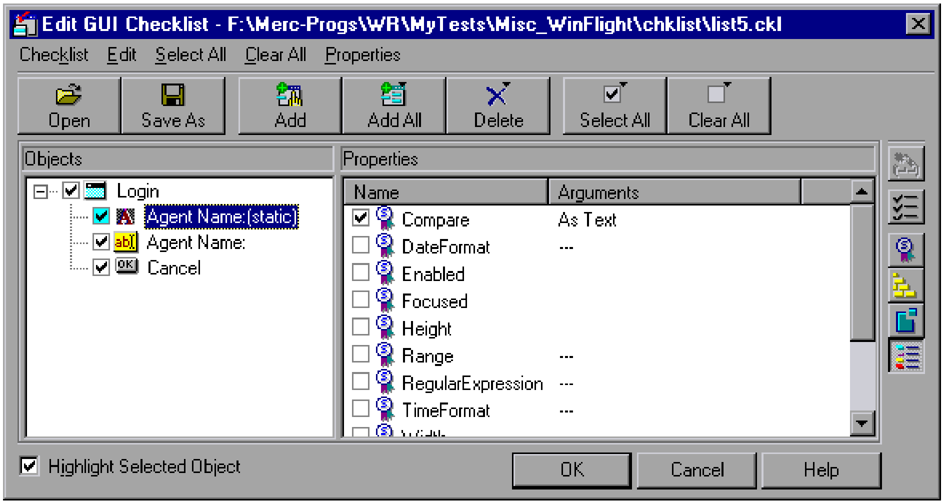
Button	Description
EXP	**Edit Expected Value** enables you to edit the expected value of the selected property. For more information, see “Editing the Expected Value of a Property” on page 219.
	Specify Arguments enables you to specify the arguments for a check on the selected property. For more information, see “Specifying Arguments for Property Checks” on page 213.
	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.
S	**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 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.

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 **Insert** > **Edit GUI Checklist**.



- ☑ Login
  - ☑ Agent Name:(static)
  - ☑ Agent Name:
  - ☑ Cancel

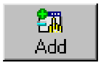
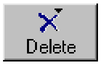
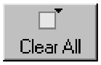
	Name	Arguments
☑	Compare	As Text
□	DateFormat	---
□	Enabled	
□	Focused	
□	Height	
□	Range	---
□	RegularExpression	---
□	TimeFormat	---
□	Width	

☑ Highlight Selected Object

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 checkmark 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 opens an existing GUI checklist.
	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** adds an object to your GUI checklist.
	Add All adds all objects or all menus in a window to your GUI checklist.
	**Delete** deletes the specified object, or all objects that appear in the GUI checklist.
	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.
	**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 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*.

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 213.
	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.
	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 621.

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 **Insert** > **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 **Insert** > **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.

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.

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 213.

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
- 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 215.

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 25, “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.

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 **Insert** > **GUI Checkpoint** > **For Multiple Objects** or **Insert** > **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 ellipsis 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.

---

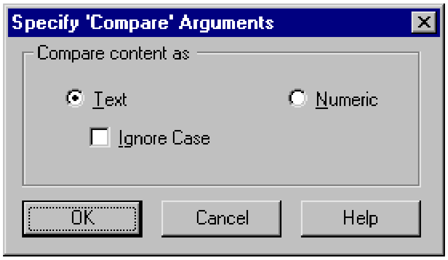
5 Specify the arguments in the dialog box that opens. For example, for a Date Format check, specify the date format. For information on specifying arguments for a particular property check, see the relevant section below.

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.

Compare Property Check

Checks the contents of the edit class or static_text class object (default check). Opens the Specify 'Compare' Arguments dialog box.



Specify 'Compare' Arguments

Compare content as

● Text ○ Numeric

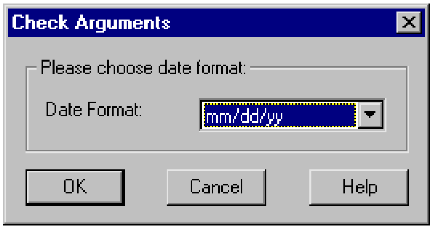
□ Ignore Case

- ➤ 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 Format: mm/dd/yy

Date Formats

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

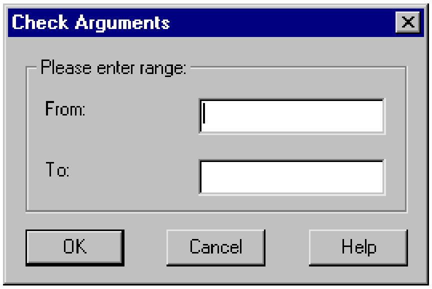
mm/dd/yy	09/24/04
dd/mm/yy	24/09/04
dd/mm/yyyy	24/09/2004
yy/dd/mm	04/24/09
dd.mm.yy	24.09.04
dd.mm.yyyy	24.09.2004
dd-mm-yy	24-09-04
dd-mm-yyyy	24-09-2004
yyyy-mm-dd	2004-09-24
Day, Month dd, yyyy	Friday (or Fri), September (or Sept) 24, 2004
dd Month yyyy	24 September 2004
Day dd Month yyyy	Friday (or Fri) 24 September (or Sept) 2004

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

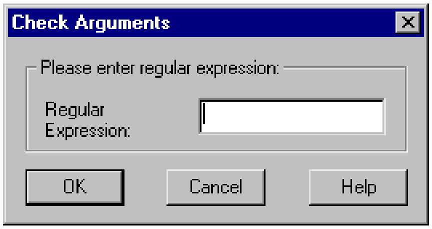
Checks that the contents of the edit class or static_text class object are within the specified range. In the Check Arguments dialog box, specify the lower limit in the top edit field, and the upper limit in the bottom edit field.

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



RegularExpression Property Check

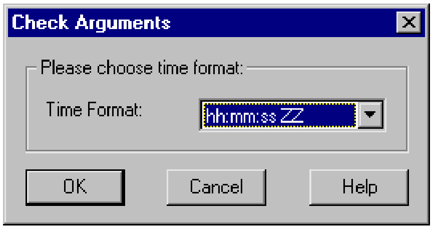
Checks that the string in the edit class or static_text class object meets the requirements of the regular expression. In the Check Arguments dialog box, enter a string into the Regular Expression box. You do not need to precede the regular expression with an exclamation point. For more information, see Chapter 25, "Using Regular Expressions."



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.



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

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.

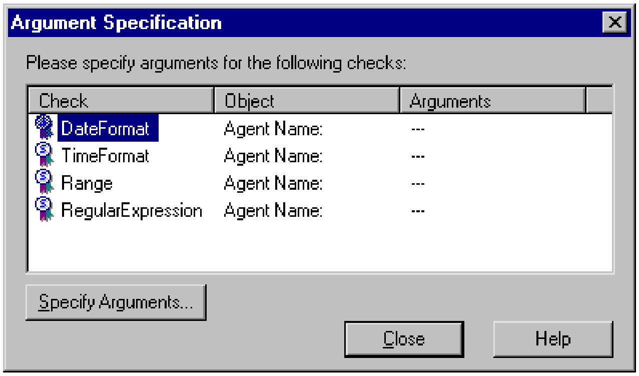
Specifying Arguments for One Property Check

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 the 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:



Please specify arguments for the following checks:

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

The *property check* appears in the **Check** column. The *logical name* of the object appears in the **Object** column. An ellipsis appears 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.

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 221.

To edit the expected value of an object property:

1 Confirm that the object for which you want to edit an expected value is displayed in your application.

Note: If the object is not displayed, WinRunner cannot display the expected value of the object's properties in the Check GUI or Create GUI Checkpoint dialog box.

---

2 If the Check GUI dialog box or the Create GUI Checkpoint dialog box is not already open, choose **Insert > 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.

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

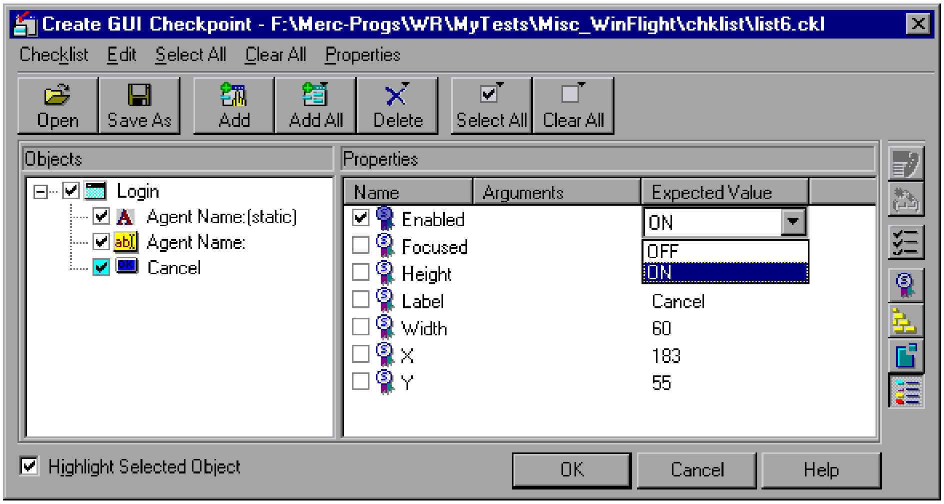
4 In the **Properties** pane, select the property whose expected value you want to edit.

5 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:



☑ Login
- ☑ Agent Name:(static)
- ☑ Agent Name:
- ☑ Cancel

Name	Arguments	Expected Value
☑ Enabled		ON ▾ OFF ON
□ Focused		
□ Height		
□ Label		Cancel
□ Width		60
□ X		183
□ Y		55

☑ Highlight Selected Object

6 Edit the expected value of the property, as desired.

7 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.

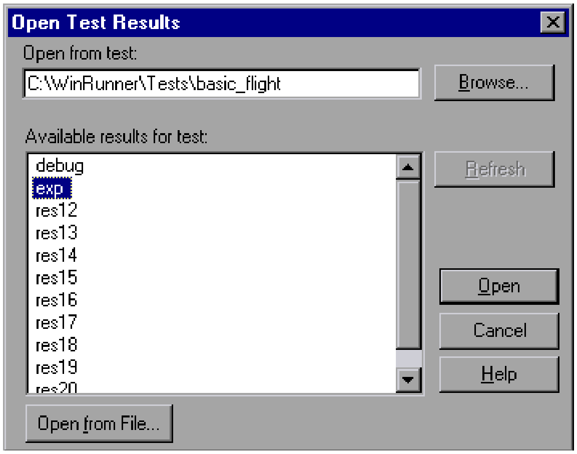
To modify the expected results for an existing GUI checkpoint:

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

The WinRunner Test Results window opens.

2 Display the expected results:

➤ In the Unified report view—Click the **Open** button or choose **File** > **Open**. The Open Test Results dialog box opens. Select **exp** and click **Open**.

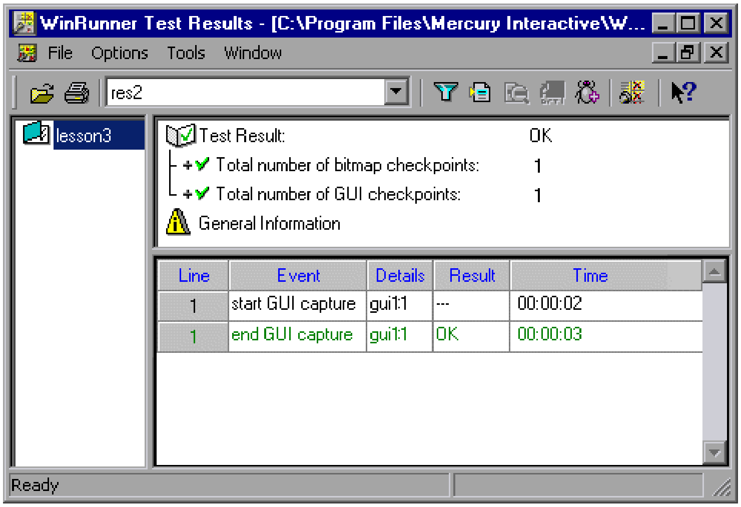


Open Test Results

Open from test: C:\WinRunner\Tests\basic_flight

Available results for test: exp

➤ In the WinRunner report view—Select **exp** in the Results location box.


WinRunner Test Results - [C:\Program Files\Mercury Interactive\W...
File Options Tools Window
res2
lesson3
Ready

Test Result: OK

- Total number of bitmap checkpoints: 1
- Total number of GUI checkpoints: 1

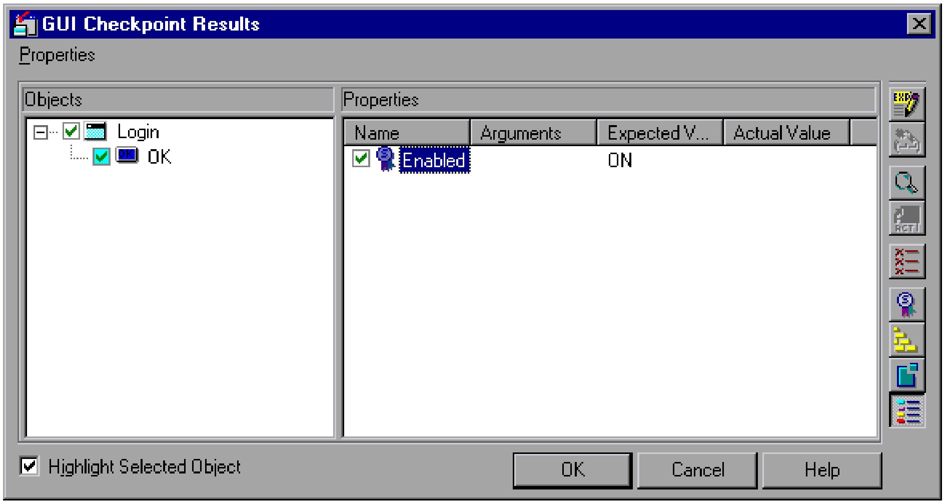
General Information

Line	Event	Details	Result	Time
1	start GUI capture	gui1:1	---	00:00:02
1	end GUI capture	gui1:1	OK	00:00:03

3 Locate the GUI checkpoint by looking for **end GUI capture** events.

Note: If you are working in the WinRunner report view, you can use the **Show TSL** button to open the test script to the highlighted line number.

4 Select and display **end GUI capture** entry. 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.

Notes: 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 219.

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 in the WinRunner Report View" on page 683.

For more information on working in the Test Results window, see Chapter 34, "Analyzing Test Results."

13

Working with Web Objects

When you load WinRunner with WebTest add-in support, WinRunner can record and run Context Sensitive operations on the Web (HTML) objects in your Web site in Netscape and Internet Explorer.

Using the WebTest add-in, you can also view the properties of Web objects, retrieve information about the Web objects in your Web site, and create checkpoints on Web objects to check the functionality of your Web site.

Note: You can also use the AOL browser to record and run tests on Web objects in your site, but you cannot record or run objects on browser elements, such as the Back, Forward, and Navigate buttons.

This chapter describes:

About Working with Web Objects

When you create tests using the WebTest add-in, WinRunner recognizes Web objects such as: frames, text links, images, tables, and Web form objects. Each object has a number of different properties. You can use these properties to identify objects, retrieve and check property values and perform Web functions.

You can also check that your Web site works as expected. For example, you can check the structure or content of frames, tables, and cells, the URL of links, the source and type of images, the color or font of text links, and more.

Note: Before you open your browser to begin testing your Web site, you must first start WinRunner with the WebTest add-in loaded. For more information, see "Loading WinRunner Add-Ins" on page 20.

Viewing Recorded Web Object Properties

You can use the **Recorded** tab of the GUI Spy to see the properties and property values that WinRunner records for the selected object just as you do for any Windows object.

To view recorded Web object properties:

1. Start WinRunner.
2. Open your Web browser.

Note: You must start WinRunner with the WebTest add-in loaded before you open your Web browser.

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

4 Select **Hide WinRunner** if you want to hide the WinRunner window (but not the GUI Spy) while you spy on the objects in your Web site.

5 Click **Spy** and point to an object in your Web page. The object is highlighted and the Window name, object name, and the recorded properties and values are displayed.

6 To capture an object description in the GUI Spy dialog box, point to an object and press the STOP softkey. (The default softkey combination is LEFT CTRL + F3.)

For more information on the GUI Spy, see “Viewing GUI Object Properties,” on page 34.

Notes:

The **All Standard** tab of the GUI Spy does not display additional (not recorded) properties of Web objects. For a list of properties associated with each Web object, see Using Web Object Properties in Your Tests.

The GUI Map Configuration tool does not support configuring all Web objects. You can use the GUI Map Configuration tool to modify how WinRunner recognizes Web objects with a window handle (HWND), such as *html_frame*, *html_edit*, *html_check_button*, *html_combobox*, *html_listbox*, *html_radio_button*, and *html_push_button*. You cannot use the GUI Map Configuration tool to modify how WinRunner recognizes Web-oriented objects such as *html_text_link* and *html_rect*. To modify how WinRunner recognizes these Web objects, you can use the GUI map configuration functions, such as **set_record_attr**, and **set_record_method**.

For more information on the GUI Map Configuration tool, see Chapter 9, “Configuring the GUI Map.” For information about the GUI map configuration functions, refer to the *TSL Reference*.

Using Web Object Properties in Your Tests

In order to create checkpoints, write statements using descriptive programming, and to take advantage of some TSL functions (such as **web_obj_get_info** and **_web_set_tag_attr**), you need to know the properties that you can use with each Web object.

This section lists and defines the properties available for each Web object including:

- Using Properties for Web Objects
- Using Properties for Frame Objects
- Using Properties for Web Images
- Using Properties for Text Links
- Using Properties for Web Tables and Table Cells
- Using Properties for Form Objects including: Radio Buttons, Check Boxes, Edit Boxes, List and Combo Boxes, and Web Buttons

For more information on checking Web objects, see "Checking Web Objects," on page 239.

For more information on descriptive programming, see "Using Descriptive Programming," on page 541.

For more information on **web_obj_get_info** and other functions that may be useful for testing a Web site, refer to the *TSL Reference*.

Using Properties for Web Objects

The following object properties are common to all Web objects except Web frames (html_frame class):

Property Name	Description
attribute/<prop_name>	Enables you to access the specified internal property of the object. For more information, see "Using attribute/<prop_name> Notation," on page 230.
bgcolor	The object's background color.
class	The WinRunner class of the object.
class_name	The object's class as it appears in the HTML.
color	The object's color.
current_bgcolor	The background color property for the element as defined by the current style. Supported only in Internet Explorer.
current_color	The color property for the element as defined by the current style. Supported only in Internet Explorer.
focused	Indicates whether the object has the focus. **Possible values: 1:** True **0:** False
height	The object's height (in pixels).
html_id	The object's HTML identifier.
inner_html	The HTML code contained between the object's start and end tags.
inner_text	The text contained between the object's start and end tags.
outer_html	The object's HTML code and its content. Supported only in Internet Explorer.

Property Name	Description
source_index	The selector value that WinRunner assigns to the object to indicate the order in which the object's HTML tag appears in the source code relative to other HTML tags. **Starting value = 0.** Supported only in Internet Explorer.
tag_name	The object's HTML tag.
tag_name	The object's HTML tag.
visible	Indicates whether the object is visible. **Possible values:** **1:** True **0:** False
width	The object's width (in pixels).

Using attribute/<prop_name> Notation

You can use the **attribute/<prop_name>** notation to identify a Web object according to its internal (user-defined) properties.

For example, suppose a Web page has the same company logo image in two places on the page:

```
<IMG src="logo.gif" LogoID="122">
<IMG src="logo.gif" LogoID="123">
```

You could identify the image that you want to click using descriptive programming by including the user-defined **LogoID** property in the object description as follows:

```
web_image_click("{class: object, MSW_class: html_rect, attribute/logoID: 123}" ,
164 , 253 );
```

For more information about descriptive programming, see "Using Descriptive Programming," on page 541.

Setting the Property to Use for the Logical Name of an Object Class

Each Web object class has a default property defined, whose value is used as the logical name of the object. You can change the default logical name property for a Web object class using the **_web_set_tag_attr** function.

If you want to use a user-defined property for the logical name of an object, you can use the **attribute/<prop_name>** notation in your **_web_set_tag_attr** statement.

For example, suppose you have the following source code in a Web page:

```
<input type="text" name="InputName1" maxlength="20" size="20" value="name"
MyAttr="Your Name">
<input type="text" name="InputName2" maxlength="20" size="20" value="name"
MyAttr="My Name">
```

By default, WinRunner would use the name attribute of the text box (InputName1 or InputName2 in the above example) as the logical name. To instruct WinRunner to use the value of the **MyAttr** property as the logical name, use the following line:

```
_web_set_tag_attr("html_edit", "attribute/MyAttr");
```

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

Using Properties for Frame Objects

The following object properties can be used when working with objects from the **html_frame** MSW class:

Property Name	Description
frame_title	The frame's title.
html_id	The frame's HTML identifier. Not supported in Netscape 4.x.

Property Name	Description
name	The WinRunner name for the object. This is the value that WinRunner uses as the logical name of the object. The value of this property is taken from the frame’s **name** property if it exists. If not, it uses the frame’s **title** property if it exists. Otherwise it uses the frame’s **url** property.
page_title	The title of the page contained in the frame.
url	The URL of the frame.

Using Properties for Web Images

In addition to the properties supported for all objects, the following properties can be used when working with objects from the **html_rect** MSW class:

Property Name	Description
alt	The object's tooltip text.
element_name	The name property specified within the <IMG> tag.
file_name	The file name of the object (without the path).
name	The WinRunner name for the object. This is the value that WinRunner uses as the logical name of the object. The value of this property is taken from the image’s **alt** property if it exists. If not, it uses the image's **name** property if it exists. Otherwise it uses the filename from the image's **src** property.
src	The object's source location (the full path).
type	The image type. **Possible values: Server side** **Client side** **Plain**

Using Properties for Text Links

In addition to the properties supported for all objects, the following properties can be used when working with objects from the **html_text_link** MSW class:

Property Name	Description
currrent_font	The font property for the link as defined by the style.
element_name	The name property specified within the <A HREF> tag.
font	The link's font.
text	The text associated with the link.
url	The URL of the link.

Using Properties for Web Tables

When working with tables, you can perform functions on table objects or cell objects.

Tables

In addition to the properties supported for all objects, the following properties can be used when working with objects from the **html_table** MSW class:

Property Name	Description
columns	The number of columns in the table.
name	The WinRunner name for the object. This is the value that WinRunner uses as the logical name of the object. The value of this property is taken from the first object in the table that has a **name** property.
rows	The number of rows in the table.

Property Name	Description
table_index	The selector value indicating the order in which the table appears in the source code relative to other tables on the page. **Starting value = 0.**
text	The text contained in the table.

Table Cells

In addition to the properties supported for all objects, the following properties can be used when working with objects from the **html_cell** MSW class:

Property Name	Description
col	The table column in which the cell is located. The first column in the table is 1.
row	The table row in which the cell is located. The first row in the table is 1.
table_index	The selector indicating the order in which the cell's table appears in the source code relative to other tables on the page. **Starting value = 0.**
text	The text contained in the cell.

Using Properties for Form Objects

When working with Web forms, you can perform functions on radio buttons, check boxes, edit boxes, list boxes, combo boxes, and buttons.

Radio Buttons

In addition to the properties supported for all objects, the following properties can be used when working with objects from the **html_radio_button** MSW class:

Property Name	Description
checked	Indicates whether or not the radio button is selected. **Possible values:** **1:** True **0:** False
element_name	The name property specified within the <input> tag.
enabled	Indicates whether or not the radio button is enabled. **Possible values:** **1:** True **0:** False
name	The WinRunner name for the object. This is the value that WinRunner uses as the logical name of the object. The value of this property is taken from the radio button's **name** property.
part_value	The button's attached text. Supported only in Internet Explorer.
value	The button's html value (label).

Check Boxes

In addition to the properties supported for all objects, the following properties can be used when working with objects from the **html_check_button** MSW class:

Property Name	Description
checked	Indicates whether or not the check box is selected. **Possible values:** **1:** True **0:** False
element_name	The name property specified within the <INPUT> tag.
enabled	Indicates whether or not the check box is enabled. **Possible values:** **1:** True **0:** False
name	The WinRunner name for the object. This is the value that WinRunner uses as the logical name of the object. The value of this property is taken from the check box’s **name** property.
part_value	The check box's value (label). Supported only in Internet Explorer.
value	The check box's value (label).

Edit Boxes

In addition to the properties supported for all objects, the following properties can be used when working with objects from the **html_edit** MSW class:

Property Name	Description
cols	The width of the edit box (in columns).
element_name	The name property specified within the <INPUT> tag.
enabled	Indicates whether or not the check box is enabled. **Possible values:** **1:** True **0:** False

Property Name	**Description**
kind	The type of edit box. **Possible values: single-line** **multi-line**
name	The WinRunner name for the object. This is the value that WinRunner uses as the logical name of the object. The value of this property is taken from the edit box's **name** property.
rows	The height of the edit box (in rows).
type	The object's type as defined in the HTML tag. For example: <input type=text>

List and Combo Boxes

In addition to the properties supported for all objects, the following properties can be used when working with objects from the **html_listbox** and **hmtl_combobox** MSW classes:

Property Name	**Description**
element_name	The name property specified within the <SELECT> tag.
is_multiple	Indicates whether the list offers a multiple selection option. **Possible values:** **1:** True **0:** False
name	The WinRunner name for the object. This is the value that WinRunner uses as the logical name of the object. The value of this property is taken from the list's **name** property.
selection	The items that are selected in the list (separated by ;).

Web Buttons

In addition to the properties supported for all objects, the following properties can be used when working with the **html_push_button** MSW class:

Property Name	Description
element_name	The name property specified within the <input> tag.
enabled	Indicates whether or not the button is enabled. **Possible values:** **1:** True **0:** False
name	The WinRunner name for the object. This is the value that WinRunner uses as the logical name of the object. The value of this property is taken from the button's **value** property if it exists. If not, it uses the button's **innertext** property if it exists. Otherwise it uses the button's **name** property.
part_value	The value of the button's "value" property if the HTML tag for the is <INPUT>. The value of the button's "innertext" property if the HTML tag for the button is <BUTTON>. Supported only in Internet Explorer.
value	The button's value (label).

Checking Web Objects

You can use GUI checkpoints in your test scripts to help you check the behavior of Web objects in your Web site. You can check frames, tables, cells, links, and images on a Web page for differences between test runs. You can define GUI checkpoints according to default properties recommended by WinRunner, or you can define custom checks by selecting other properties. For general information on GUI checkpoints, see Chapter 12, "Checking GUI Objects."

You can also add text checkpoints in your test scripts to read and check text in Web objects and in areas of the Web page.

You can create checkpoints for:

- Checking Standard Frame Properties
- Checking the Object Count in Frames
- Checking the Structure of Frames, Tables, and Cells
- Checking the Content of Frames, Cells, Links, or Images
- Checking the Number of Columns and Rows in a Table
- Checking the URL of Links
- Checking Source or Type of Images and Image Links
- Checking Color or Font of Text Links
- Checking Broken Links
- Checking Links and Images in a Frame
- Checking the Text Content of Tables
- Checking Cells in a Table
- Checking Text

Checking Standard Frame Properties

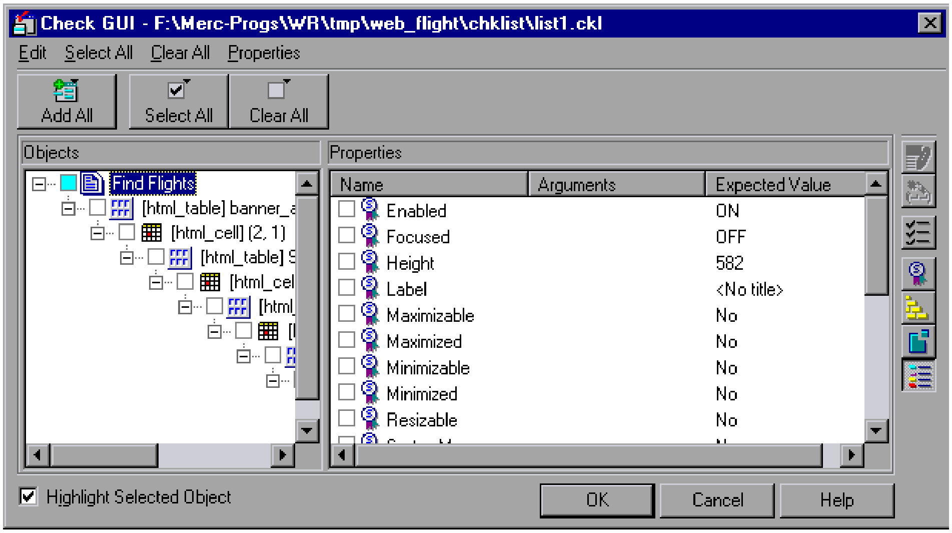
You can create a GUI checkpoint to check standard properties of a frame.

To check standard frame properties:

1 Choose **Insert** > **GUI Checkpoint** > **For Object/Window**.

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

2 Double-click an object on your Web page. The Check GUI dialog box opens, and the object is highlighted.



Check GUI - F:\Merc-Progs\WR\tmp\web_flight\chklist\list1.ckl

Find Flights
[html_table] banner_
[html_cell] (2, 1)
[html_table] S
[html_cel
[html

Name	Arguments	Expected Value
□ Enabled		ON
□ Focused		OFF
□ Height		582
□ Label		<No title>
□ Maximizable		No
□ Maximized		No
□ Minimizable		No
□ Minimized		No
□ Resizable		No

☑ Highlight Selected Object

3 In the **Objects** column, make sure that the frame is selected.

The Properties column indicates the available standard properties and the default check for that frame.

4 In the **Properties** column, choose the properties you want WinRunner to check.

You can check the following standard properties:

- **Enabled** checks whether the frame can be selected.
- **Focused** checks whether keyboard input will be directed to this frame.
- **Label** checks the frame's label.

- **Minimizable** and **Maximizable** check whether the frame can be minimized or maximized.
- **Minimized** and **Maximized** check whether the frame is minimized or maximized.
- **Resizable** checks whether the frame can be resized.
- **SystemMenu** checks whether the frame has a system menu.
- **Width** and **Height** check the frame's width and height, in pixels.
- **X** and **Y** check the x and y coordinates of the top left corner of the frame.

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

WinRunner captures the object information and stores it in the test's expected results folder. The WinRunner window is restored and a checkpoint appears in your test script as a **win_check_gui** statement. For more information on the **win_check_gui** function, refer to the *TSL Reference* (**Help** > **TSL Reference**).

Checking the Object Count in Frames

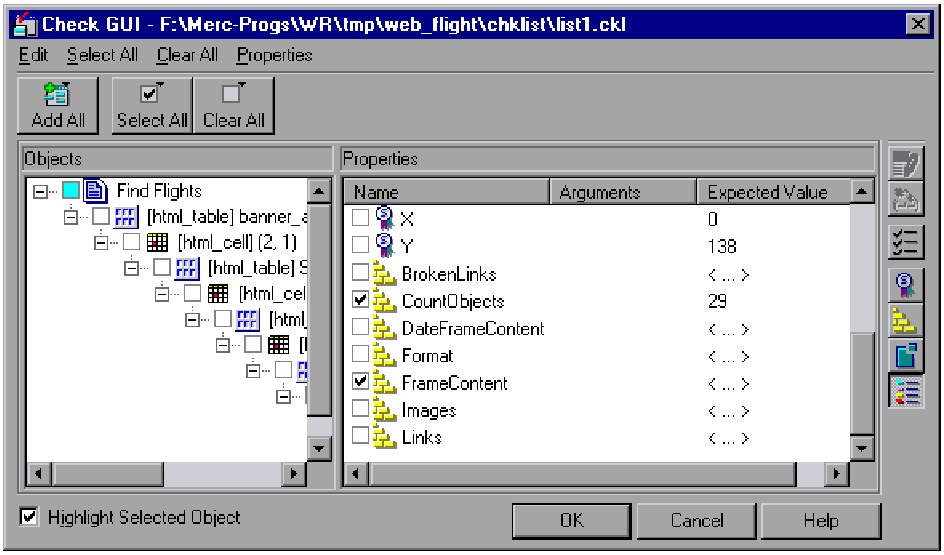
You can create a GUI checkpoint to check the number of objects in a frame.

To check the object count in a frame:

1 Choose **Insert** > **GUI Checkpoint** > **For Object/Window**.

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

2 Double-click an object on your Web page. The **Check GUI** dialog box opens, and the object is highlighted.



Name	Arguments	Expected Value
□ X		0
□ Y		138
□ BrokenLinks		< ... >
☑ CountObjects		29
□ DateFrameContent		< ... >
□ Format		< ... >
☑ FrameContent		< ... >
□ Images		< ... >
□ Links		< ... >

3 In the **Objects** column, make sure that the frame is selected.

The Properties column indicates the properties available for you to check.

4 In the **Properties** column, select the **CountObjects** check box.

5 To edit the expected value of the property, highlight **CountObjects**.

Click the **Edit Expected Value** button, or double-click the value in the **Expected Value** column to edit it. A spin box opens.

Enter the expected number of objects.

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

WinRunner captures the object information and stores it in the test's expected results folder. The WinRunner window is restored and a checkpoint appears in your test script as a **win_check_gui** statement. For more information on the **win_check_gui** function, refer to the *TSL Reference* (**Help > TSL Reference**).

Checking the Structure of Frames, Tables, and Cells

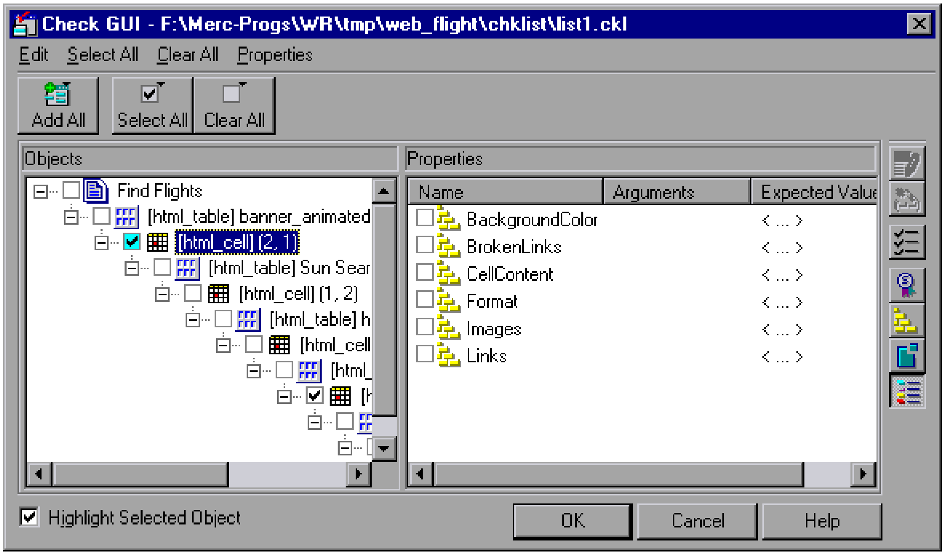
You can create a GUI checkpoint to check the structure of frames, tables, and cells on a Web page.

To check the structure of a frame, table, or cell:

1 Choose **Insert** > **GUI Checkpoint** > **For Object/Window**.

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

2 Double-click an object on your Web page. The Check GUI dialog box opens, and the object is highlighted.



Name	Arguments	Expected Value
□ BackgroundColor		< ... >
□ BrokenLinks		< ... >
□ CellContent		< ... >
□ Format		< ... >
□ Images		< ... >
□ Links		< ... >

3 In the **Objects** column, select an object.

The Properties column indicates the properties available for you to check.

4 In the **Properties** column, select the **Format** check box.

5 To edit the expected value of the property, highlight **Format**.

Click the **Edit Expected Value** button, or double-click the value in the **Expected Value** column to edit it. A text file opens in Notepad describing the structure of the frame, table, or cell.

Modify the expected structure.

Save the text file and close Notepad.

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

WinRunner captures the object information and stores it in the test's expected results folder. The WinRunner window is restored and a checkpoint appears in your test script as an **obj_check_gui** or **win_check_gui** statement. For more information on the **obj_check_gui** and **win_check_gui** functions, refer to the *TSL Reference* (**Help** > **TSL Reference**).

Checking the Content of Frames, Cells, Links, or Images

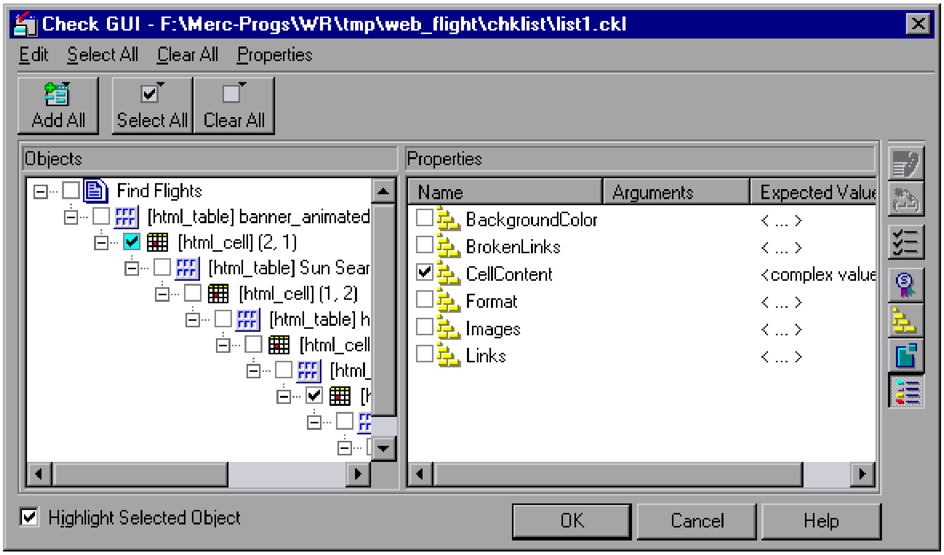
You can create a GUI checkpoint to check the content of a frame, cell, text link, image link, or an image. To check the content of a table, see "Checking the Text Content of Tables" on page 255.

To check content:

1 Choose **Insert** > **GUI Checkpoint** > **For Object/Window**.

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

2 Double-click an object on your Web page. The Check GUI dialog box opens, and the object is highlighted.



Name	Arguments	Expected Value
□ BackgroundColor		< ... >
□ BrokenLinks		< ... >
☑ CellContent		<complex value
□ Format		< ... >
□ Images		< ... >
□ Links		< ... >

3. In the **Objects** column, select an object (frame, cell, text link, image link, or an image).

   The Properties column indicates the properties available for you to check.

4. In the **Properties** column, select one of the following checks:

   - If your object is a frame, select the **FrameContent** check box.
   - If your object is a cell, select the **CellContent** check box.
   - If your object is a text link, select the **Text** check box.
   - If your object is an image link, select the **ImageContent** check box.
   - If your object is an image, select the **ImageContent** check box.

5. To edit the expected value of a the property, highlight a property.

   Note that you cannot edit the expected value of the **ImageContent** property.

6. Click the **Edit Expected Value** button, or double-click the value in the **Expected Value** column to edit it.

   - For the **FrameContent** property, an editor opens.
   - For the **CellContent** property, an editor opens.
   - For the **Text** property, an edit box opens.

7. Modify the expected value.

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

   WinRunner captures the object information and stores it in the test's expected results folder. The WinRunner window is restored and a checkpoint appears in your test script as an **obj_check_gui** or **win_check_gui** statement. For more information on the **obj_check_gui** and **win_check_gui** functions, refer to the *TSL Reference* (**Help > TSL Reference**).

Checking the Number of Columns and Rows in a Table

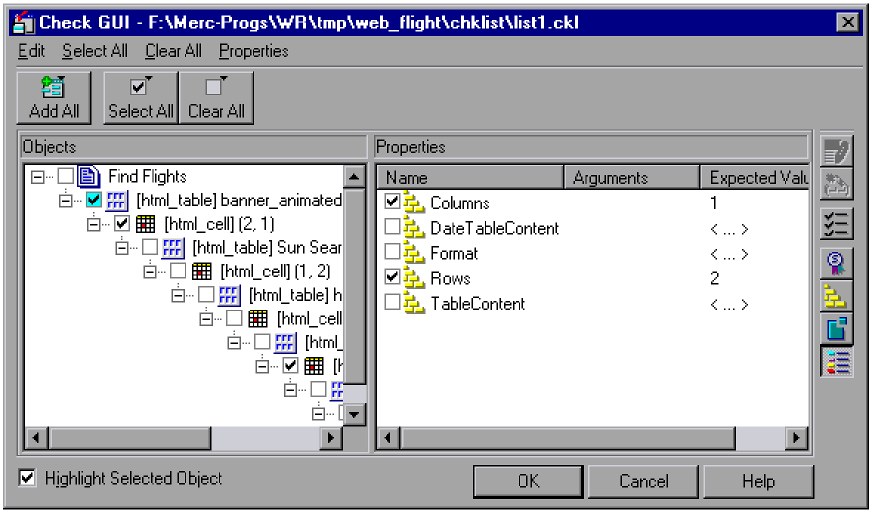
You can create a GUI checkpoint to check the number of columns and rows in a table.

To check the number of columns and rows in a table:

1 Choose **Insert** > **GUI Checkpoint** > **For Object/Window**.

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

2 Double-click a table on your Web page. The Check GUI dialog box opens, and the object is highlighted.



Check GUI - F:\Merc-Progs\WR\tmp\web_flight\chklist\list1.ckl

□ Find Flights
☑ [html_table] banner_animated
☑ [html_cell] (2, 1)
□ [html_table] Sun Sear
□ [html_cell] (1, 2)
□ [html_table] h
□ [html_cell
□ [html_
☑ [h

Name	Arguments	Expected Valu
☑ Columns		1
□ DateTableContent		< ... >
□ Format		< ... >
☑ Rows		2
□ TableContent		< ... >

☑ Highlight Selected Object

3 In the **Objects** column, make sure the table is selected.

The Properties column indicates the properties available for you to check.

4 In the **Properties** column, select the **Columns** and/or **Rows** check box.

5 To edit the expected value of a property, highlight **Columns** or **Rows**.

Click the **Edit Expected Value** button, or double-click the value in the **Expected Value** column to edit it. A spin box opens.

Edit the expected value of the property, as desired.

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

WinRunner captures the object information and stores it in the test's expected results folder. The WinRunner window is restored and a checkpoint appears in your test script as an **obj_check_gui** or **win_check_gui** statement. For more information on the **obj_check_gui** and **win_check_gui** functions, refer to the *TSL Reference* (**Help > TSL Reference**).

Checking the URL of Links

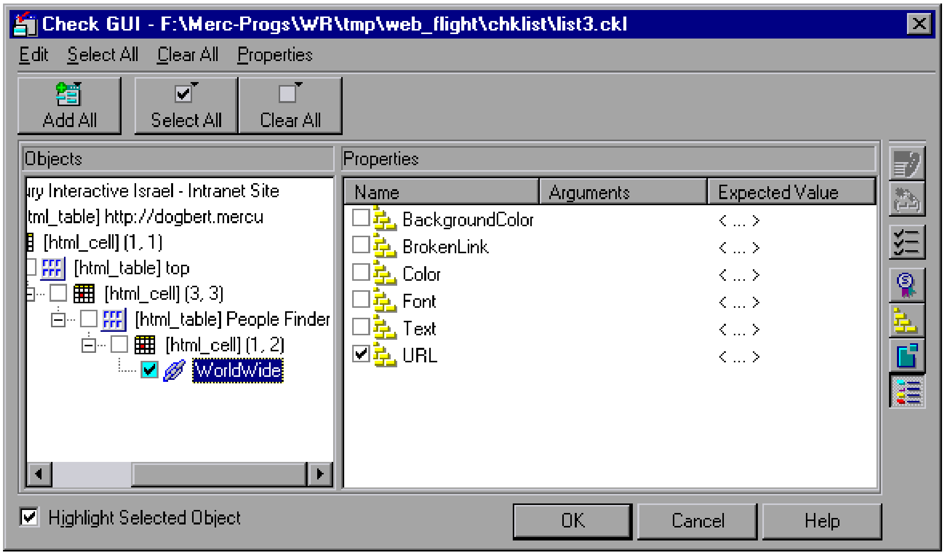
You can create a GUI checkpoint to check the URL of a text link or an image link in your Web page.

To check the URL of a link:

1 Choose **Insert > GUI Checkpoint > For Object/Window.**

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

2 Double-click a text link on your Web page. The Check GUI dialog box opens, and the object is highlighted.



ry Interactive Israel - Intranet Site
tml_table] http://dogbert.mercu
[html_cell] (1, 1)
□ [html_table] top
□ [html_cell] (3, 3)
□ [html_table] People Finder
□ [html_cell] (1, 2)
☑ WorldWide

Name	Arguments	Expected Value
□ BackgroundColor		< ... >
□ BrokenLink		< ... >
□ Color		< ... >
□ Font		< ... >
□ Text		< ... >
☑ URL		< ... >

☑ Highlight Selected Object

3 In the **Objects** column, make sure that link is selected.

The **Properties** column indicates the properties available for you to check.

4 In the **Properties** column, select **URL** to check address of the link.

5 To edit the expected value of the URL property, highlight **URL**.

Click the **Edit Expected Value** button, or double-click the value in the **Expected Value** column to edit it. An edit box opens.

Edit the expected value.

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

WinRunner captures the object information and stores it in the test's expected results folder. The WinRunner window is restored and a checkpoint appears in your test script as an **obj_check_gui** statement. For more information on the **obj_check_gui** function, refer to the *TSL Reference* (**Help** > **TSL Reference**).

Checking Source or Type of Images and Image Links

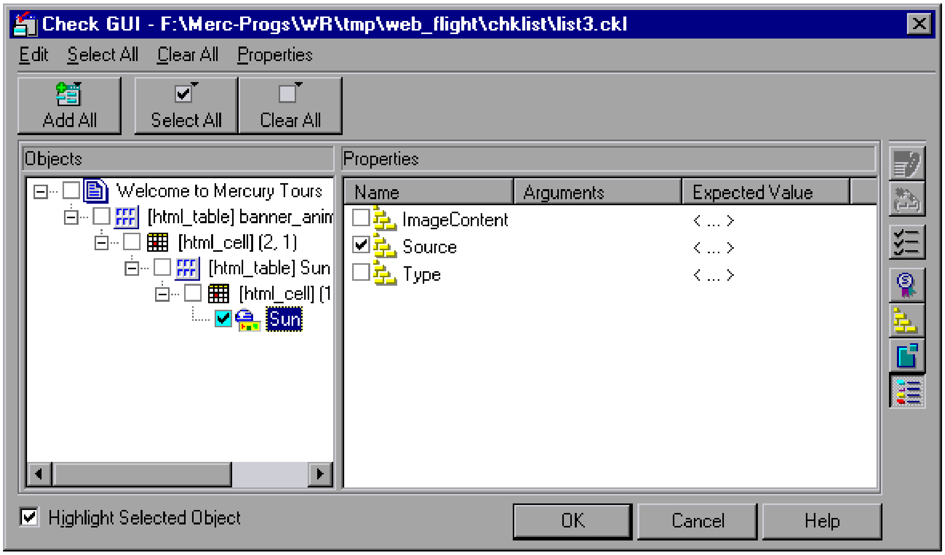
You can create a GUI checkpoint to check the source and the image type of an image or an image link in your Web page.

To check the source or type of an image or an image link:

1 Choose **Insert** > **GUI Checkpoint** > **For Object/Window**.

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

2 Double-click an image or image link on your Web page. The Check GUI dialog box opens, and the object is highlighted.



□ Welcome to Mercury Tours
□ [html_table] banner_anim
□ [html_cell] (2, 1)
□ [html_table] Sun
□ [html_cell] (1
☑ Sun

Name	Arguments	Expected Value
□ ImageContent		< ... >
☑ Source		< ... >
□ Type		< ... >

☑ Highlight Selected Object

3 In the **Objects** column, make sure that the image or the image link is selected.

The Properties column indicates the properties available for you to check.

4 In the Properties column, select a property check.

- **Source** indicates the location of the image.
- **Type** indicates whether the object is a plain image, an image link, or an image map.

5 To edit the expected value of the property, highlight a property.

Click the **Edit Expected Value** button, or double-click the value in the **Expected Value** column to edit it. An edit box opens.

Edit the expected value.

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

WinRunner captures the object information and stores it in the test's expected results folder. The WinRunner window is restored and a checkpoint appears in your test script as an **obj_check_gui** statement. For more information on the **obj_check_gui** function, refer to the *TSL Reference* (**Help** > **TSL Reference**).

Checking Color or Font of Text Links

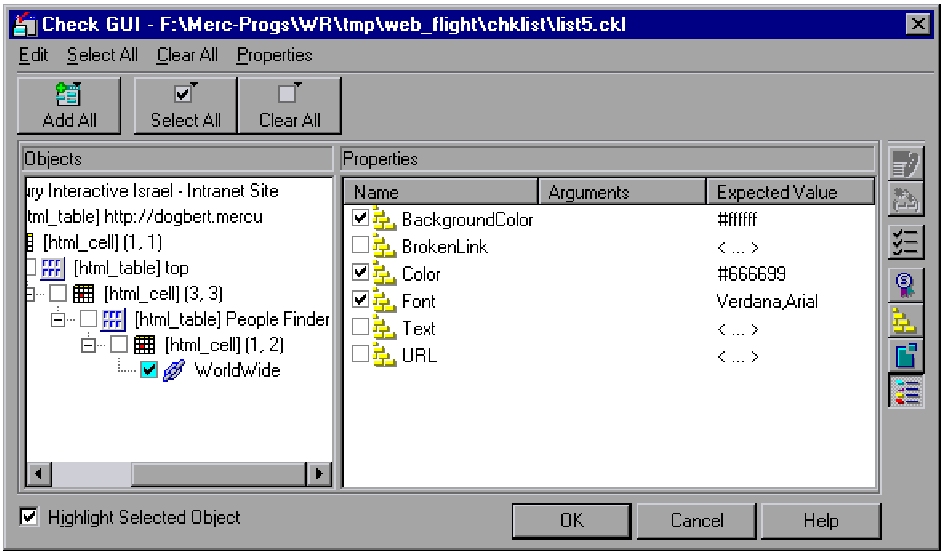
You can create a GUI checkpoint to check the color and font of a text link in your Web page.

To check the color or font of a text link:

1 Choose **Insert** > **GUI Checkpoint** > **For Object/Window**.

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

2 Double-click a text link on your Web page. The Check GUI dialog box opens, and the object is highlighted.



Check GUI - F:\Merc-Progs\WR\tmp\web_flight\chklist\list5.ckl

ıry Interactive Israel - Intranet Site
tml_table] http://dogbert.mercu
[html_cell] (1, 1)
□ [html_table] top
□ [html_cell] (3, 3)
□ [html_table] People Finder
□ [html_cell] (1, 2)
☑ WorldWide

Name	Arguments	Expected Value
☑ BackgroundColor		#ffffff
□ BrokenLink		< ... >
☑ Color		#666699
☑ Font		Verdana,Arial
□ Text		< ... >
□ URL		< ... >

☑ Highlight Selected Object

3 In the **Objects** column, make sure that the text link is selected.

The Properties column indicates the properties available for you to check.

4. In the Properties column, select a property check.

   - **BackgroundColor** indicates the background color of a text link.
   - **Color** indicates the foreground color of a text link.
   - **Font** indicates the font of a text link.

5. To edit the expected value of a property, highlight a property.

   Click the **Edit Expected Value** button, or double-click the value in the **Expected Value** column to edit it. A box opens.

   Edit the expected value.

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

   WinRunner captures the object information and stores it in the test's expected results folder. The WinRunner window is restored and a checkpoint appears in your test script as an **obj_check_gui** statement.

   For more information on the **obj_check_gui** function, refer to the *TSL Reference* (**Help** > **TSL Reference**).

Checking Broken Links

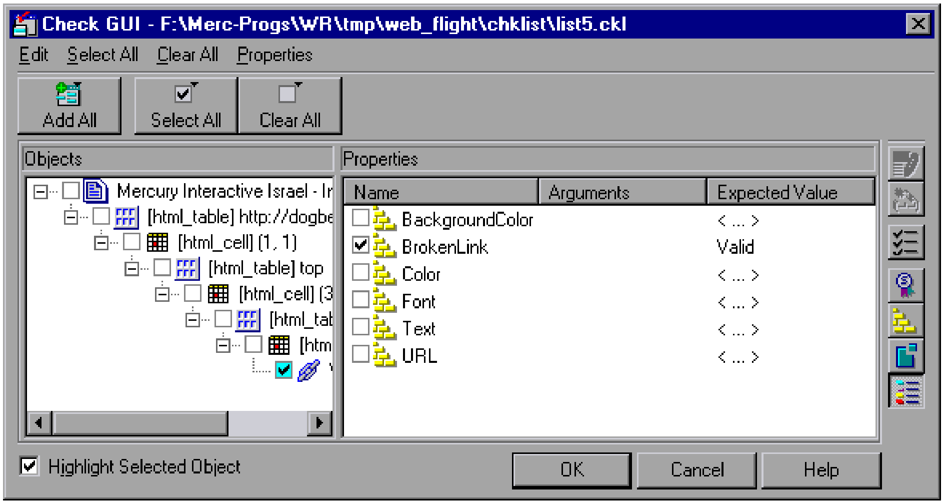
You can create a checkpoint to check whether a text link or an image link is active. You can create a checkpoint to check a single broken link or all the broken links in a frame.

To check a single broken link:

1. Choose **Insert** > **GUI Checkpoint** > **For Object/Window**.

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

2 Double-click a link on your Web page. The Check GUI dialog box opens, and the object is highlighted.



Objects:

- ☐ Mercury Interactive Israel - Ir
  - ☐ [html_table] http://dogbe
    - ☐ [html_cell] (1, 1)
      - ☐ [html_table] top
        - ☐ [html_cell] (3
          - ☐ [html_tal
            - ☐ [htm
              - ☑

Name	Arguments	Expected Value
☐ BackgroundColor		< ... >
☑ BrokenLink		Valid
☐ Color		< ... >
☐ Font		< ... >
☐ Text		< ... >
☐ URL		< ... >

☑ Highlight Selected Object

3 In the **Objects** column, make sure that the link is selected.

The Properties column indicates the properties available for you to check.

4 In the **Properties** column, select the **BrokenLink** check box.

5 To edit the expected value of the property, highlight **BrokenLink**.

Click the **Edit Expected Value** button, or double-click the value in the **Expected Value** column to edit it. A combo box opens.

Select **Valid** or **NotValid**. Valid indicates that the link is active, and NotValid indicates that the link is broken.

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

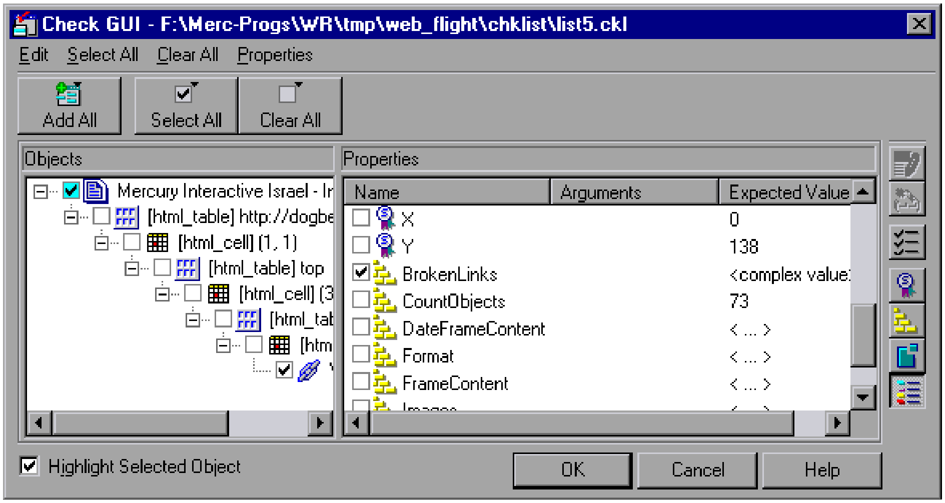
WinRunner captures the object information and stores it in the test's expected results folder. The WinRunner window is restored and a checkpoint appears in your test script as an **obj_check_gui** or **win_check_gui** statement. For more information on the **obj_check_gui** and **win_check_gui** function, refer to the *TSL Reference* (**Help > TSL Reference**).

To check all broken links in a frame:

1 Choose **Insert** > **GUI Checkpoint** > **For Object/Window.**

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

2 Double-click an object on your Web page. The Check GUI dialog box opens, and an object is highlighted.



Check GUI - F:\Merc-Progs\WR\tmp\web_flight\chklist\list5.ckl

Objects

- ☑ Mercury Interactive Israel - Ir
  - □ [html_table] http://dogbe
    - □ [html_cell] (1, 1)
      - □ [html_table] top
        - □ [html_cell] (3
          - □ [html_tal
            - □ [htm
              - ☑

Properties

Name	Arguments	Expected Value
□ X		0
□ Y		138
☑ BrokenLinks		<complex value:
□ CountObjects		73
□ DateFrameContent		< ... >
□ Format		< ... >
□ FrameContent		< ... >
□ Images		< ... >

☑ Highlight Selected Object

3 In the **Objects** column, make sure that frame is selected.

The Properties column indicates the properties available for you to check.

4 In the **Properties** column, select the **BrokenLinks** check box.

5 To edit the expected value of the property, highlight **BrokenLinks**.

Click the **Edit Expected Value** button, or double-click the value in the **Expected Value** column to edit it. The Edit Check dialog box opens.

You can specify which links to check, and which verification method and verification type to use. You can also edit the expected data. For additional information on using this dialog box, see “Checking Cells in a Table” on page 257.

When you are done, click **OK** to save and close the Edit Check dialog box. The Check GUI dialog box is restored.

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

WinRunner captures the object information and stores it in the test's expected results folder. The WinRunner window is restored and a checkpoint appears in your test script as a **win_check_gui** statement. For more information on the **win_check_gui** function, refer to the *TSL Reference* (**Help > TSL Reference**).

Checking Links and Images in a Frame

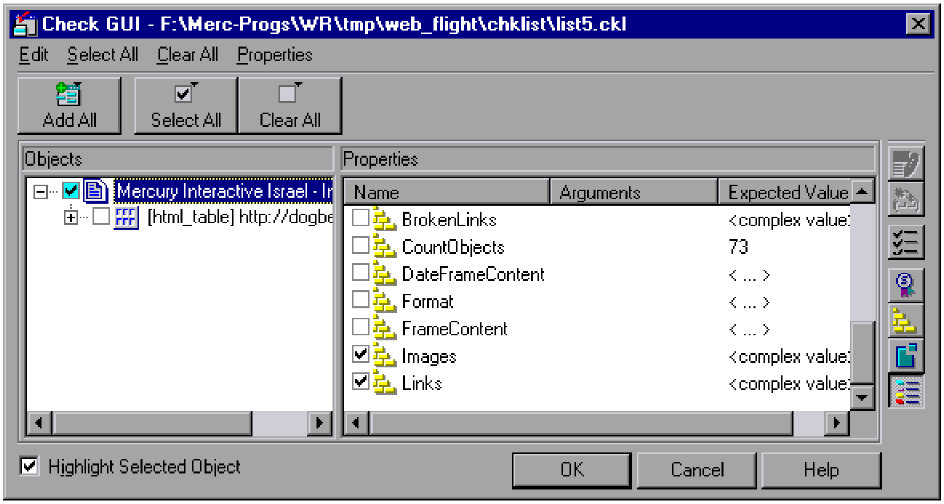
You can create a checkpoint to check image links, text links and images in a frame.

To check links and images in a frame:

1 Choose **Insert > GUI Checkpoint > For Object/Window.**

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

2 Double-click an object on your Web page. The Check GUI dialog box opens, and an object is highlighted.



Check GUI - F:\Merc-Progs\WR\tmp\web_flight\chklist\list5.ckl

Objects:

- ☑ Mercury Interactive Israel - Ir
- ☐ [html_table] http://dogbe

Name	Arguments	Expected Value
☐ BrokenLinks		<complex value
☐ CountObjects		73
☐ DateFrameContent		< ... >
☐ Format		< ... >
☐ FrameContent		< ... >
☑ Images		<complex value
☑ Links		<complex value

☑ Highlight Selected Object

3 In the **Objects** column, make sure that frame object is selected.

The Properties column indicates the properties available for you to check.

4 In the **Properties** column, select one of the following checks:

- ➤ To check images or image links, select the **Images** check box.
- ➤ To check text links, select the **Links** check box.

5 To edit the expected value of the property, highlight **Images**.

Click the **Edit Expected Value** button, or double-click the value in the **Expected Value** column to edit it. The **Edit Check** dialog box opens.

You can specify which images or links to check in the table, and which verification method and verification type to use. You can also edit the expected data. For additional information on using this dialog box, see “Checking Cells in a Table” on page 257.

When you are done, click **OK** to save and close the Edit Check dialog box. The Check GUI dialog box is restored.

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

WinRunner captures the object information and stores it in the test's expected results folder. The WinRunner window is restored and a checkpoint appears in your test script as a **win_check_gui** statement. For more information on the **win_check_gui** function, refer to the *TSL Reference* (**Help** > **TSL Reference**).

Checking the Text Content of Tables

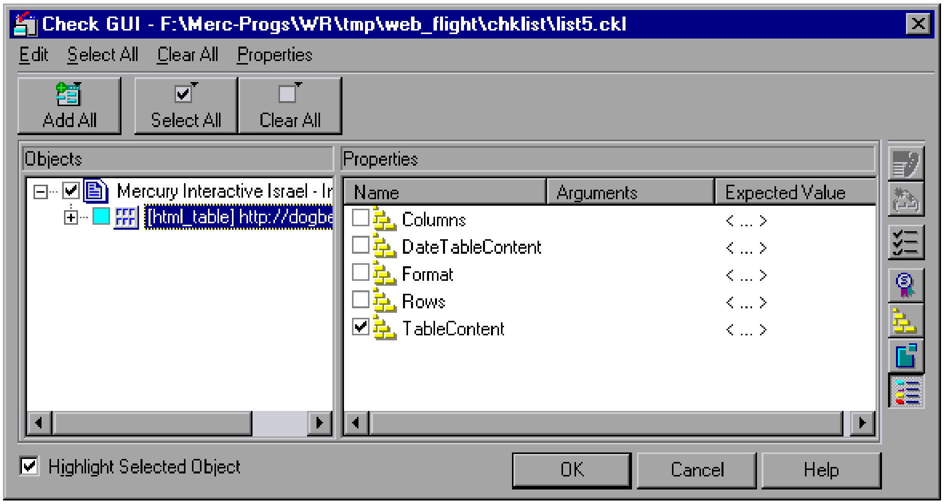
You can create a checkpoint to check the text content of a table.

To check the content of a table:

1 Choose **Insert** > **GUI Checkpoint** > **For Object/Window**.

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

2 Double-click a table on your Web page. The Check GUI dialog box opens, and an object is highlighted.



Check GUI - F:\Merc-Progs\WR\tmp\web_flight\chklist\list5.ckl

Objects:

- ⊟ ☑ Mercury Interactive Israel - Ir
  - ⊞ [html_table] http://dogbe

Properties:

Name	Arguments	Expected Value
□ Columns		< ... >
□ DateTableContent		< ... >
□ Format		< ... >
□ Rows		< ... >
☑ TableContent		< ... >

☑ Highlight Selected Object

3 In the **Objects** column, make sure that the table is selected.

The Properties column indicates the properties available for you to check.

4 In the **Properties** column, select the **TableContent** check box.

5 To edit the expected value of the property, highlight **TableContent**.

Click the **Edit Expected Value** button, or double-click the value in the **Expected Value** column to edit it. The **Edit Check** dialog box opens.

You can specify which column or rows to check in the table, and which verification method and verification type to use. You can also edit the expected data. For additional information on using this dialog box, see "Checking Cells in a Table" on page 257.

When you are done, click **OK** to save and close the Edit Check dialog box. The Check GUI dialog box is restored.

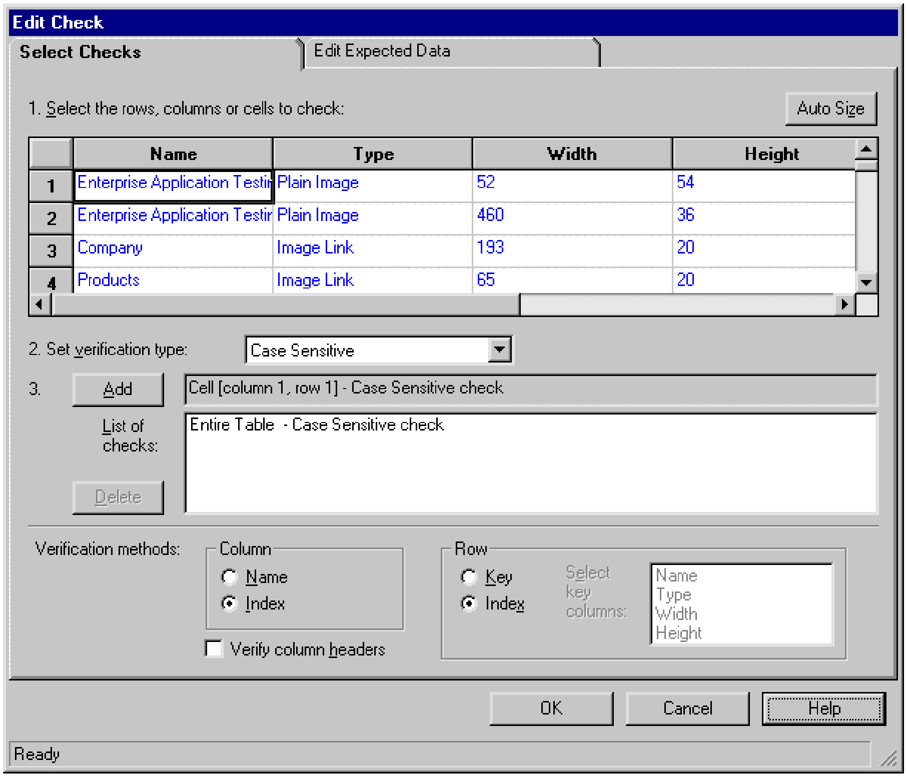
6 Click **OK** to close the Check GUI dialog box.

WinRunner captures the object information and stores it in the test's expected results folder. The WinRunner window is restored and a checkpoint appears in your test script as a **win_check_gui** statement.

For more information on the **win_check_gui** function, refer to the *TSL Reference* (**Help > TSL Reference**).

Checking Cells in a Table

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.



1. Select the rows, columns or cells to check:

	Name	Type	Width	Height
1	Enterprise Application Testir	Plain Image	52	54
2	Enterprise Application Testir	Plain Image	460	36
3	Company	Image Link	193	20
4	Products	Image Link	65	20

2. Set verification type: Case Sensitive

3. Cell [column 1, row 1] - Case Sensitive check

List of checks: Entire Table - Case Sensitive check

Verification methods:
Column: ○ Name ● Index
□ Verify column headers
Row: ○ Key ● Index
Select key columns: Name, Type, Width, Height

Ready

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 260.

Specifying which Cells to Check

The **List of checks** box 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**.

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 262.

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

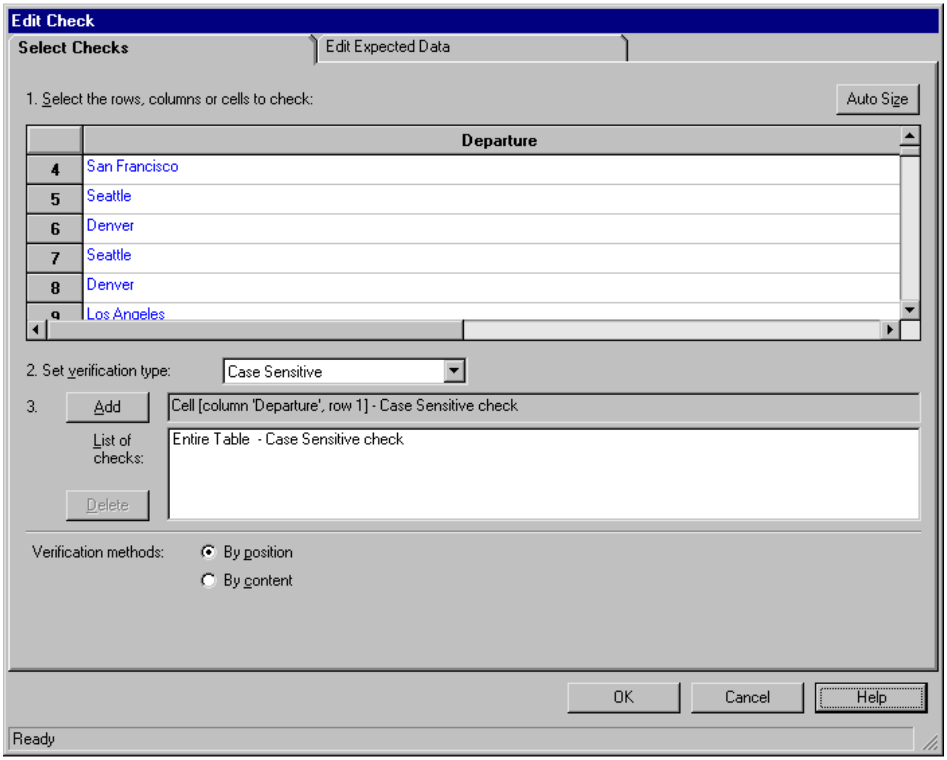
- Column:
  - **Name:** WinRunner looks for the selection according to the column names. A shift in the position of the columns within the table does not result in a mismatch.
  - **Index:** WinRunner looks for the selection according to the index, or position, of the columns. A shift in the position 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 258. 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 data in the key column(s) specified in the **Select key columns** list box. For example, you could tell WinRunner to identify the second row in the table on page 263 based on the arrival time for that row. A shift in the position of the rows does not result in a mismatch. If the key selection does not uniquely identify a row, WinRunner checks the first matching row. You can use more than one key column to uniquely identify the row.

Note: If the value of a cell in one or more of the key columns changes, WinRunner will not be able to identify the corresponding row, and a check of that row will fail with a "Not Found" error. If this occurs, select a different key column or use the Index verification method.

- **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 methods** 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.



	Departure
4	San Francisco
5	Seattle
6	Denver
7	Seattle
8	Denver
9	Los Angeles

2. Set verification type: Case Sensitive

3. Cell [column 'Departure', row 1] - Case Sensitive check

List of checks: Entire Table - Case Sensitive check

Verification methods: ● By position ○ By content

- **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.

Note: This option causes a mismatch on any string that does not begin with a number. A string starting with 'e' is translated into a number.

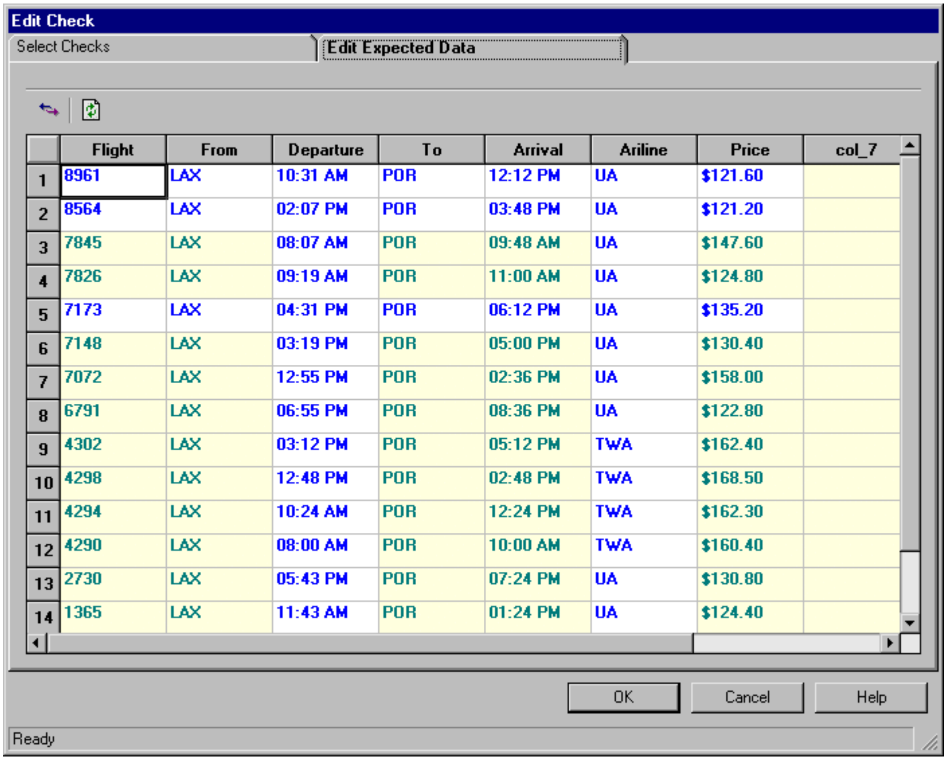
- **Case Sensitive Ignore Spaces:** WinRunner checks the data in the cell according to case and content, ignoring differences in spaces. WinRunner reports any differences in case or content as a mismatch.
- **Case Insensitive Ignore Spaces:** WinRunner checks the content in the cell according to content,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



	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	

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.

Checking Text

You can use text checkpoints in your test scripts to read and check text in Web objects and in areas of the Web page. While creating a test, you point to an object or a frame containing text. WebTest 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 a text string or all the text from a Web object or frame, using **web_obj_get_text** or **web_frame_get_text**
- check that a text string exists in a Web object or frame, using **web_obj_text_exists** or **web_frame_text_exists**

Reading All the Text in a Frame or an Object

You can read all the visible text in a frame or an object using **web_obj_get_text** or **web_frame_get_text**.

To read all the text in a frame or an object:

1. Choose **Insert** > **Get Text** > **From Object/Window**.

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

2. Click the Web object or the frame.

   WinRunner captures the text in the object and a **web_obj_get_text** or a **web_frame_get_text** statement is inserted in your test script.

Note: When the WebTest add-in is not loaded, or when a non-Web object is selected, WinRunner generates a **win_get_text** or **obj_get_text** statement in your test script. For more information on the **_get_text** functions, refer to the *TSL Reference* (**Help** > **TSL Reference**). For more information on checking text in a non-Web object, see Chapter 19, "Checking Text."

Reading a Text String from a Frame or an Object

You can read a text string from a frame or an object using the **web_obj_get_text** or **web_frame_get_text** function.

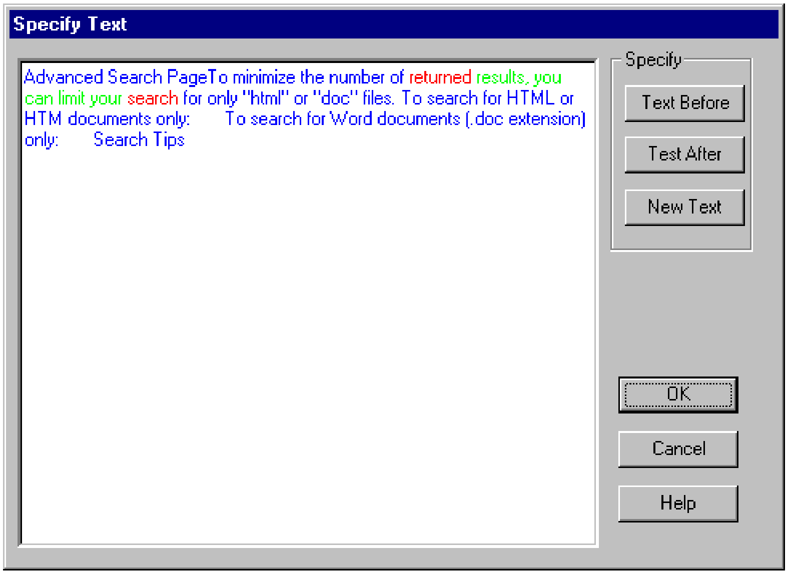
To read a text string from a frame or an object:

1 Choose **Insert** > **Get Text** > **From Selection (Web only)**.

WinRunner is minimized, the mouse pointer becomes a pointing hand, and a Help window opens.

2 Highlight the text string to be read.

3 On the highlighted text string, right-click the mouse button to capture the string. The Specify Text dialog box opens.



Advanced Search PageTo minimize the number of returned results, you can limit your search for only "html" or "doc" files. To search for HTML or HTM documents only: To search for Word documents (.doc extension) only: Search Tips

The text string to be read is displayed in green. The red text that is displayed on the left and right of your selection, defines the bounds of the string.

4 You can modify your text selections.

- ➤ To modify your highlighted text selection, highlight a new text string and click **New Text**.

  Your new text selection is displayed in green. The text that appears before and after your text string is displayed in red.

- ➤ To modify the red text string that appears to the left of your selection, highlight a new text string and click **Text Before**.

- ➤ To modify the red text string that appears to the right of your selection, highlight a new text string and click **Text After**.

5 Click **OK** to close the Specify Text dialog box.

The WinRunner window is restored and a **web_obj_get_text** or a **web_frame_get_text** statement is inserted in your test script.

Checking that a Text String Exists in a Frame or an Object

You can check whether a text string exists in an object or a frame using **web_obj_text_exists** or **web_frame_text_exists**.

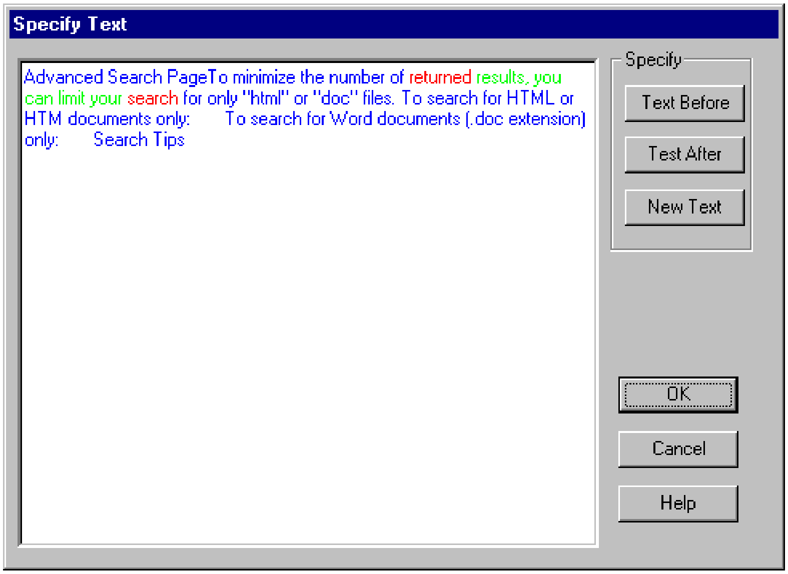
To check that a text string exists in a frame or an object:

1 Choose **Insert** > **Get Text** > **Web Text Checkpoint**.

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

2 Highlight the text string to be checked.

3 On the highlighted text string, right-click the mouse button to capture the string. The Specify Text dialog box opens.



Advanced Search PageTo minimize the number of returned results, you can limit your search for only "html" or "doc" files. To search for HTML or HTM documents only: To search for Word documents (.doc extension) only: Search Tips

The text string to be checked is displayed in green. The red text that is displayed on the left and right of your selection defines the bounds of the string.

4 You can modify your text selections.

➤ To modify your highlighted text selection, highlight a new text string and click **New Text**.

Your new text selection is displayed in green. The text that is displayed before and after your text string is displayed in red.

➤ To modify the red text string that is displayed to the left of your selection, highlight a new text string and click **Text Before**.

➤ To modify the red text string that is displayed to the right of your selection, highlight a new text string and click **Text After**.

5 Click **OK** to close the Specify Text dialog box.

The WinRunner window is restored and a **web_obj_text_exists** or a **web_frame_text_exists** statement is inserted in your test script.

Note: After you run your test, a **check_text** statement is displayed in your Test Results window.

14

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 and other applications.

This chapter describes:

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 supported controls, as well as check their properties.

WinRunner supports all standard (built-in) Visual Basic and ActiveX controls. WinRunner also offers a more customized Context Sensitive support for several ActiveX Controls. For a list of these controls, see "Supported ActiveX Controls," on page 271.

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

- install and load add-in support for ActiveX and Visual Basic controls (also known as non-agent support)
- compile a WinRunner agent into your application, and install and load add-in support for 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 12, "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 GUI Spy. In addition, you can retrieve and set the values of properties for ActiveX and Visual Basic controls using TSL functions. You can also use a TSL function to activate an ActiveX control method.

Note: If you are using non-agent support, you must start WinRunner before launching the application containing ActiveX and Visual Basic controls.

WinRunner provides special built-in support for checking Visual Basic label controls and 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 288. For information on checking the contents of an ActiveX table control, see Chapter 16, "Checking Table Contents."

Supported ActiveX Controls

WinRunner supports all ActiveX controls. WinRunner also offers a more customized Context Sensitive support for certain ActiveX Controls. The following lists summarize the controls with special support. For the latest list of supported controls and detailed ProgID and version information, refer to the *WinRunner Read Me*.

Button Objects

The following ActiveX controls are supported for button objects:

- Sheridan ActiveThreeD Control
  Sheridan Data CommandButton Control
  Sheridan OLE Data CommandButton Control

Calendar Objects

The following ActiveX controls are supported for calendar objects:

- Crescent CSCalendar Control
- Sheridan MonthView Control

Check Box Objects

The following ActiveX controls are supported for check box objects:

- Sheridan ActiveThreeD Control

Combo Box Objects

The following ActiveX controls are supported for combo box objects:

- Sheridan Data Combo Control
  Sheridan OLE Data Combo Control

Edit Objects

The following ActiveX controls are supported for edit objects:

- FarPoint InputPro Control

List Objects

The following ActiveX controls are supported for list objects:

- FarPoint ListPro Control
- Microsoft ListView Control

Menu and Toolbar Objects

The following ActiveX controls are supported for menu and toolbar objects:

- DataDynamics ActiveBar Control
- Infragistics UltraToolBar Control
- Sheridan ActiveToolBars Control
  Sheridan ActiveToolBars Plus Control

Radio Button Objects

The following ActiveX controls are supported for radio button objects:

- Sheridan ActiveThreeD Control

Radio Group Objects

The following ActiveX controls are supported for radio group objects:

- Sheridan Data Option Set Control
  Sheridan OLE Data Option Set Control

Tab Objects

The following ActiveX controls are supported for tab objects:

- Microsoft TabStrip Control
- Sheridan ActiveTabs Control

Table Objects

The following ActiveX controls are supported for ActiveX tables:

- Apex True DBGrid Control,
  Apex True OLE DBGrid Control
- FarPoint Spread Control
  FarPoint Spread (OLEDB) Control
- Infragistics UltraGrid (supported for running tests only)
- Microsoft DataBound Grid Control
  Microsoft DataGrid Control
  Microsoft FlexGrid Control
  Microsoft Grid Control
  Microsoft Hierarchical FlexGrid Control
- Sheridan Data Grid Control
  Sheridan OLE DBGrid
  Sheridan DBData Option Set
  Sheridan OLEDBData Option Set
  Sheridan DBCombo
  Sheridan OLE DBCombo
  Sheridan DBData Command
  Sheridan OLEDBData Command

Toolbar Objects

The following ActiveX controls are supported for tool bar objects:

- DataDynamics ActiveBar Control
- Microsoft Toolbar Control
- Sheridan ActiveToolBars Control
  Sheridan ActiveToolBars Plus Control

Tree Objects

The following ActiveX controls are supported for tree objects:

- Microsoft TreeView Control
- Sheridan ActiveTreeView Control

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:

- install and load add-in support for ActiveX and Visual Basic controls (also known as non-agent support)
- compile a WinRunner agent into your application, and install and load add-in support for Visual Basic controls

When you work with add-in support for ActiveX and Visual Basic controls, you can:

- record and run tests with operations on supported ActiveX and 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

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 20.

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 20.

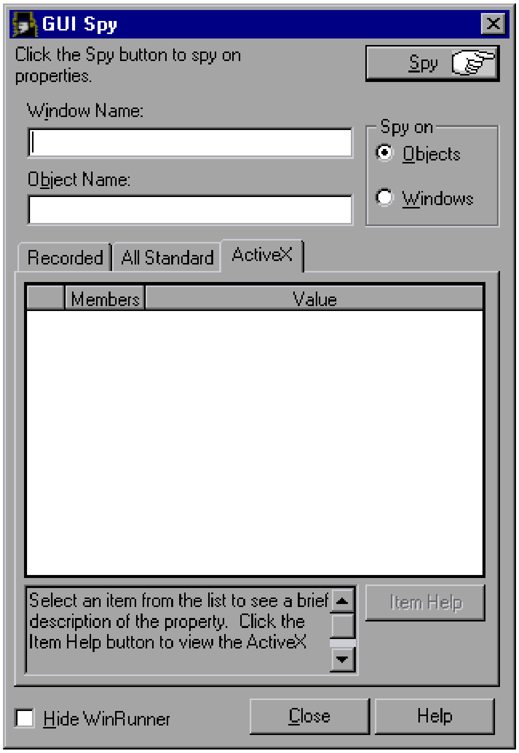
Viewing ActiveX and Visual Basic Control Properties

You use the **ActiveX** tab of the GUI Spy to see the properties, property values, and methods for an ActiveX control. You open the GUI Spy from the Tools menu. Note that in order for the GUI Spy to work on ActiveX controls, you must load the ActiveX add-in when you start WinRunner. 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 12, “Checking GUI Objects.”

To view the properties of an ActiveX or a Visual Basic control:

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

2 Click the **ActiveX** tab.



GUI Spy

Click the Spy button to spy on properties.

Spy on: ● Objects ○ Windows

Select an item from the list to see a brief description of the property. Click the Item Help button to view the ActiveX

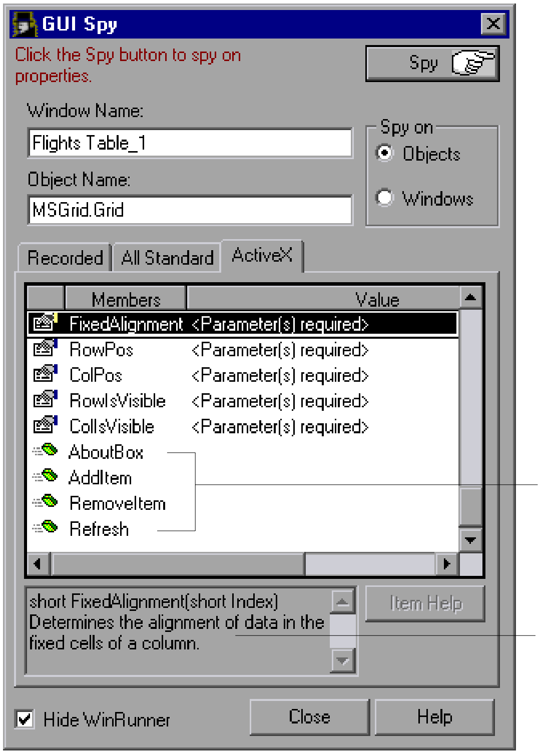
□ Hide WinRunner

3 Click **Spy** and point to an ActiveX or Visual Basic control.

The control 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 name appears in the **Object Name** box.

4 To capture an object description in the GUI Spy dialog box, point to the desired object and press the STOP softkey. (The default softkey combination is CTRL LEFT + F3.)

In the following example, pointing to the "Flights Table" in the Visual Basic sample flight application, pressing the STOP softkey, and highlighting the FixedAlignment property, displays the **ActiveX** tab in the GUI Spy as follows:



GUI Spy

Click the Spy button to spy on properties.

Window Name: Flights Table_1

Object Name: MSGrid.Grid

Spy on: ● Objects ○ Windows

Members	Value
FixedAlignment	<Parameter(s) required>
RowPos	<Parameter(s) required>
ColPos	<Parameter(s) required>
RowIsVisible	<Parameter(s) required>
ColIsVisible	<Parameter(s) required>
AboutBox	
AddItem	
RemoveItem	
Refresh	

short FixedAlignment(short Index)
Determines the alignment of data in the fixed cells of a column.

☑ Hide WinRunner

If a help file has been installed for this ActiveX control, then clicking **Item Help** displays it.

When you highlight a property, then if a description has been included for this property, it is displayed in the gray pane at the bottom.

5 Click **Close** to close the GUI Spy.

Note: When **Object Reference** appears in the **Value** column, it refers to the object's sub-objects and their properties. When **<Parameter(s) Required>** 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 "Retrieving the Value of an ActiveX or Visual Basic Control Property" on page 278 or refer to the *TSL Reference*.

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

The **ActiveX_get_info** and **ActiveX_set_info** 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 27, "Generating Functions."

Tip: You can view the properties of an ActiveX control property from the **ActiveX** tab of the GUI Spy. For additional information, see "Viewing ActiveX and Visual Basic Control Properties" on page 275.

Retrieving 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 property can have no parameters or a one or two-dimensional array. Properties can also be nested.

For an ActiveX property without parameters, the syntax is as follows:

ActiveX_get_info (*ObjectName*, *PropertyName*, *OutValue* [**,** *IsWindow*] **);**

For an ActiveX property that is a one-dimensional array, the syntax is as follows:

ActiveX_get_info (*ObjectName*, *PropertyName* **(** *X* **) ,** *OutValue*
 [**,** *IsWindow*] **);**

For an ActiveX property that is a two-dimensional array, the syntax is as follows:

ActiveX_get_info (*ObjectName*, *PropertyName* **(** *X* **,** *Y* **) ,** *OutValue*
 [**,** *IsWindow*] **);**

ObjectName The name of the ActiveX/Visual Basic control.

PropertyName Any ActiveX/Visual Basic control property.

Tip: You can use the **ActiveX** tab in the GUI Spy to view the properties of an ActiveX control.

OutValue The output variable that stores the property value.

IsWindow An indication of whether the operation is performed on a window. If it is, set this parameter to TRUE.

Note: The *IsWindow* parameter should be used only when this function is applied to a Visual Basic form to get its property or a property of its sub-object. In order to get a property of a label control you should set this parameter to TRUE. For information on retrieving label control properties, see "Working with Visual Basic Label Controls" on page 282.

Note: To get the value of nested properties, you can use any combination of indexed or non-indexed properties separated by a dot. For example:

```
ActiveX_get_info("Grid", "Cell(10,14).Text", Text);
```

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 property can have no parameters or a one or two-dimensional array. Properties can also be nested.

For an ActiveX property without parameters, the syntax is as follows:

ActiveX_set_info (*ObjectName, PropertyName, Value* [**,** *Type*
[**,** *IsWindow*]] **);**

For an ActiveX property that is a one-dimensional array, the syntax is as follows:

ActiveX_set_info (*ObjectName***,** *PropertyName* **(** *X* **) ,** *Value* [**,** *Type*
[**,** *IsWindow*]] **);**

For an ActiveX property that is a two-dimensional array, the syntax is as follows:

ActiveX_set_info (*ObjectName***,** *PropertyName* **(** *X* **,** *Y* **) ,** *Value* [**,** *Type*
[**,** *IsWindow*]] **);**

ObjectName	The name of the ActiveX/Visual Basic control.
PropertyName	Any ActiveX/Visual Basic control property.

Tip: You can use the **ActiveX** tab in the GUI Spy to view the properties of an ActiveX control.

Value	The value to be applied to the property.
Type	The value type to be applied to the property. The following types are available:

VT_I2 (short)	VT_I4 (long)	VT_R4 (float)
VT_R8 (float double)	VT_DATE (date)	VT_BSTR (string)
VT_ERROR (S code)	VT_BOOL (boolean)	VT_UI1 (unsigned char)

IsWindow	An indication of whether the operation is performed on a window. If it is, set this parameter to TRUE.

Note: The *IsWindow* parameter should be used only when this function is applied to a Visual Basic form to set its property or a property of its sub-object. In order to get a property of a label control you should set this parameter to TRUE. For information on setting label control properties, see "Working with Visual Basic Label Controls" on page 282.

Note: To set the value of nested properties, you can use any combination of indexed or non-indexed properties separated by a dot. For example:

```
ActiveX_set_info("Book", "Chapter(7).Page(2).Caption", "SomeText");
```

For more information on these functions and examples of usage, refer to the *TSL Reference*.

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 Reference*.

Working with Visual Basic Label Controls

WinRunner includes the following support for labels (static text controls) within Visual Basic applications:

- Creating GUI Checkpoints
- Retrieving Label Control Names
- Retrieving Label Properties
- Setting Label Properties

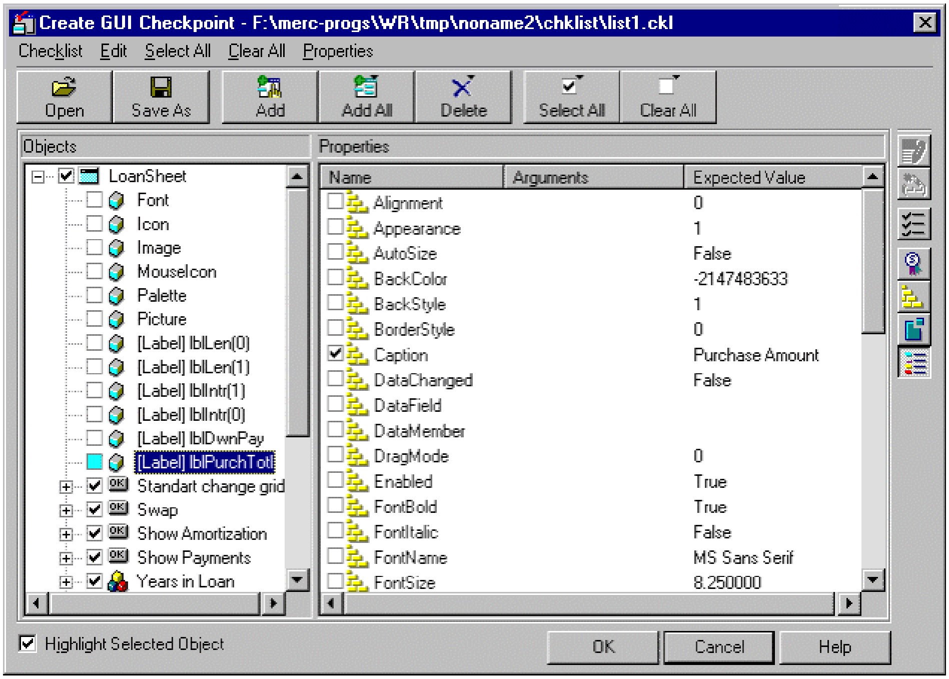
Creating GUI Checkpoints

You can create GUI checkpoints on Visual Basic label controls.

To check Visual Basic Label controls:

1. Choose **Insert** > **GUI Checkpoint** > **For Multiple Objects**. The Create GUI Checkpoint dialog box opens.
2. Click the **Add** button and click on the Visual Basic form containing Label controls.
3. The Add All dialog box opens. If you are not checking anything else in this checkpoint, you can clear the Objects check box. Click **OK**. Right-click to finish adding the objects. In the Create GUI Checkpoint dialog box, all labels are listed in the Objects pane as sub-objects of the VB form window. The names of these sub-objects are *vb_names* prefixed by the "[Label]" string.

4 When you select a label control in the Objects pane, its properties and their values are displayed in the Properties pane. The default check for the label control is the **Caption** property check. You can also select other property checks to perform.



Create GUI Checkpoint - F:\merc-progs\WR\tmp\noname2\chklist\list1.ckl

- ☑ LoanSheet
  - □ Font
  - □ Icon
  - □ Image
  - □ MouseIcon
  - □ Palette
  - □ Picture
  - □ [Label] lblLen(0)
  - □ [Label] lblLen(1)
  - □ [Label] lblIntr(1)
  - □ [Label] lblIntr(0)
  - □ [Label] lblDwnPay
  - ■ [Label] lblPurchTotl
  - ☑ Standart change grid
  - ☑ Swap
  - ☑ Show Amortization
  - ☑ Show Payments
  - ☑ Years in Loan

Name	Arguments	Expected Value
□ Alignment		0
□ Appearance		1
□ AutoSize		False
□ BackColor		-2147483633
□ BackStyle		1
□ BorderStyle		0
☑ Caption		Purchase Amount
□ DataChanged		False
□ DataField		
□ DataMember		
□ DragMode		0
□ Enabled		True
□ FontBold		True
□ FontItalic		False
□ FontName		MS Sans Serif
□ FontSize		8.250000

☑ Highlight Selected Object

Retrieving Label Control Names

You use the **vb_get_label_names** function to retrieve the list of label controls within the Visual Basic form. This function has the following syntax:

vb_get_label_names (*window*, *name_array*, *count* **);**

window The logical name of the Visual Basic form.

name_array The out parameter containing the name of the storage array.

count The out parameter containing the number of elements in the array.

This function retrieves the names of all label controls in the given form window. The names are stored as subscripts of an array.

Note: The first element in the array index is numbered 1.

For more information on this function and an example of usage, refer to the *TSL Reference*.

Retrieving Label Properties

You use the **ActiveX_get_info** function to retrieve the property value of a label control within a Visual Basic form. This function is described in "Retrieving and Setting the Values of ActiveX and Visual Basic Control Properties" on page 278.

Setting Label Properties

You use the **ActiveX_set_info** function to set the property value of the label control. This function is described in "Retrieving and Setting the Values of ActiveX and Visual Basic Control Properties" on page 278.

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 12, "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.

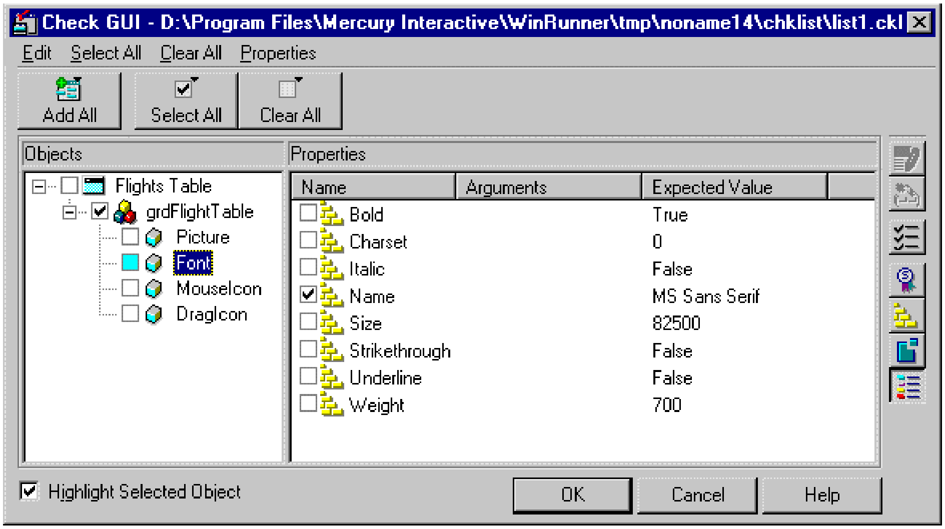
To check the sub-objects of an ActiveX or a Visual Basic control:

1 Choose **Insert** > **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.

The Check GUI dialog box opens.



Check GUI - D:\Program Files\Mercury Interactive\WinRunner\tmp\noname14\chklist\list1.ckl

Objects:

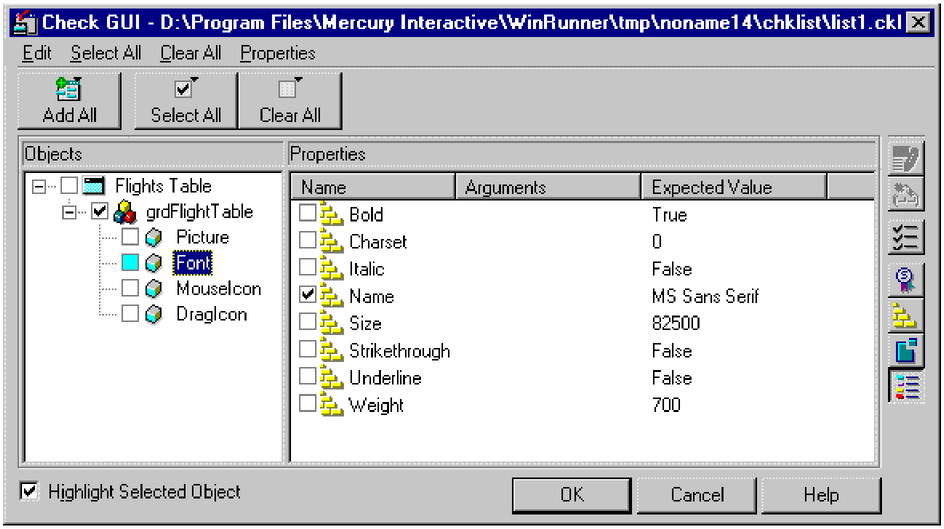
- □ Flights Table
  - ☑ grdFlightTable
    - □ Picture
    - ■ Font
    - □ MouseIcon
    - □ DragIcon

Properties:

Name	Arguments	Expected Value
□ Bold		True
□ Charset		0
□ Italic		False
☑ Name		MS Sans Serif
□ Size		82500
□ Strikethrough		False
□ Underline		False
□ Weight		700

☑ Highlight Selected Object

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.

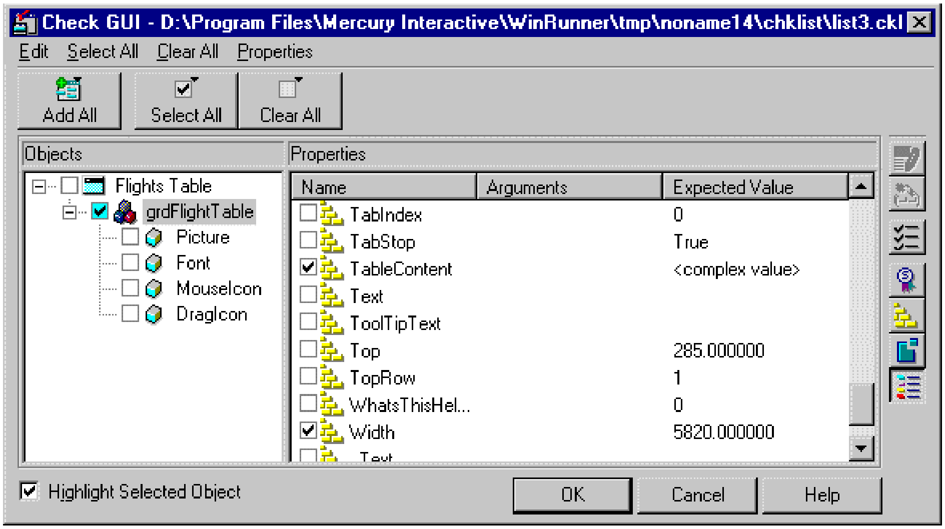


Name	Arguments	Expected Value
☐ Bold		True
☐ Charset		0
☐ Italic		False
☑ Name		MS Sans Serif
☐ Size		82500
☐ Strikethrough		False
☐ Underline		False
☐ Weight		700

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 **TableContent** properties. If you do not want to perform these checks, clear the appropriate check boxes. For information on checking table contents, see Chapter 16, "Checking Table Contents."


Check GUI - D:\Program Files\Mercury Interactive\WinRunner\tmp\noname14\chklist\list3.ckl
Edit
Select All
Clear All
Properties
Add All
Select All
Clear All
Objects
Properties
OK
Cancel
Help

- □ Flights Table
  - ☑ grdFlightTable
    - □ Picture
    - □ Font
    - □ MouseIcon
    - □ DragIcon

Name	Arguments	Expected Value
□ TabIndex		0
□ TabStop		True
☑ TableContent		<complex value>
□ Text		
□ ToolTipText		
□ Top		285.000000
□ TopRow		1
□ WhatsThisHel...		0
☑ Width		5820.000000
□ Text		

☑ Highlight Selected Object

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 12, "Checking GUI Objects,"or refer to the *TSL 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, examples of usage, and supported versions of ActiveX controls, refer to the *TSL Reference*.

	Data Bound Grid Control	FarPoint Spreadsheet Control	Microsoft FlexGrid, Grid Control	Sheridan Data Grid Control	Apex True DBGrid Control	Infragistics UltraGrid 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	+	+	+		+	+

15

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:

- About Checking PowerBuilder Applications
- Checking Properties of DropDown Objects
- Checking Properties of DataWindows
- Checking Properties of Objects within DataWindows
- Working with Computed Columns in DataWindows

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

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 16, "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 **Test** > **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 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. If you are checking, for example, a DropDownListBox, you can 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:

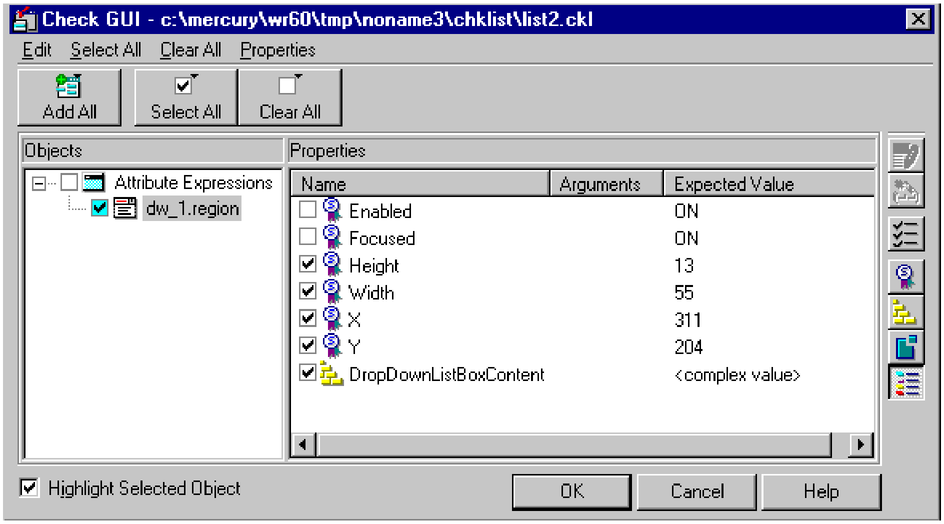
1 Choose **Test** > **Record–Context Sensitive** or click the **Record–Context Sensitive** button.

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.



Objects:

- □ Attribute Expressions
  - ☑ dw_1.region

Properties:

	Name	Arguments	Expected Value
□	Enabled		ON
□	Focused		ON
☑	Height		13
☑	Width		55
☑	X		311
☑	Y		204
☑	DropDownListBoxContent		<complex value>

☑ Highlight Selected Object

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 304.

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 Reference*.

Note: If you wish to check additional objects while performing a check on the contents, use the **Insert** > **GUI Checkpoint** > **For Multiple Objects** command (instead of the **Insert** > **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 12, "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 16, "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 **Insert** > **GUI Checkpoint** > **For Object/Window** or click the **GUI Checkpoint for Object/Window** button on the User toolbar.
2. Click in the DataWindow once.

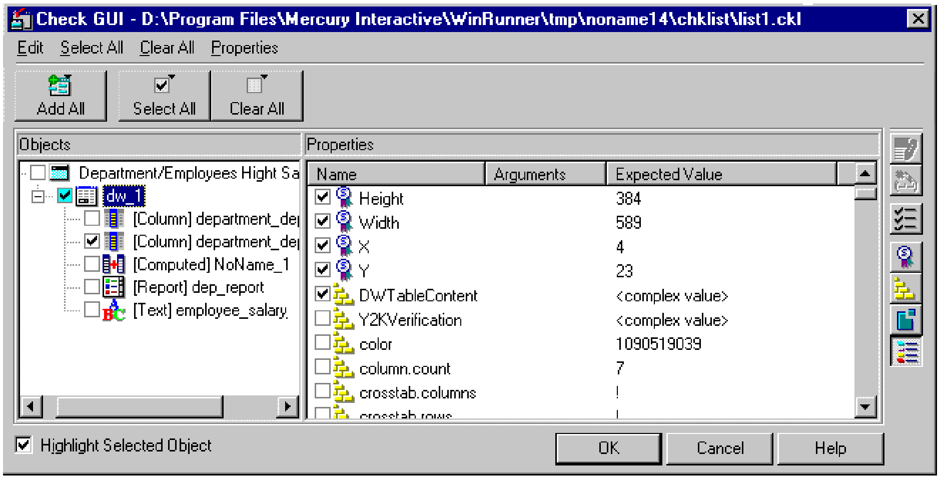
WinRunner captures the GUI information and stores it in the test's expected results folder. The WinRunner window is restored and an **obj_check_gui** statement is inserted into the test script. For more information on the **obj_check_gui** function, refer to the *TSL Reference*.

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 **Insert** > **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.



Check GUI - D:\Program Files\Mercury Interactive\WinRunner\tmp\noname14\chklist\list1.ckl

- ☐ Department/Employees Hight Sa
  - ☑ dw_1
    - ☐ [Column] department_de|
    - ☑ [Column] department_de|
    - ☐ [Computed] NoName_1
    - ☐ [Report] dep_report
    - ☐ [Text] employee_salary

Name	Arguments	Expected Value
☑ Height		384
☑ Width		589
☑ X		4
☑ Y		23
☑ DWTableContent		<complex value>
☐ Y2KVerification		<complex value>
☐ color		1090519039
☐ column.count		7
☐ crosstab.columns		!
☐ crosstab.rows		!

☑ Highlight Selected Object

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 295.

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 304.

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 Reference*.

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 12, "Checking GUI Objects."

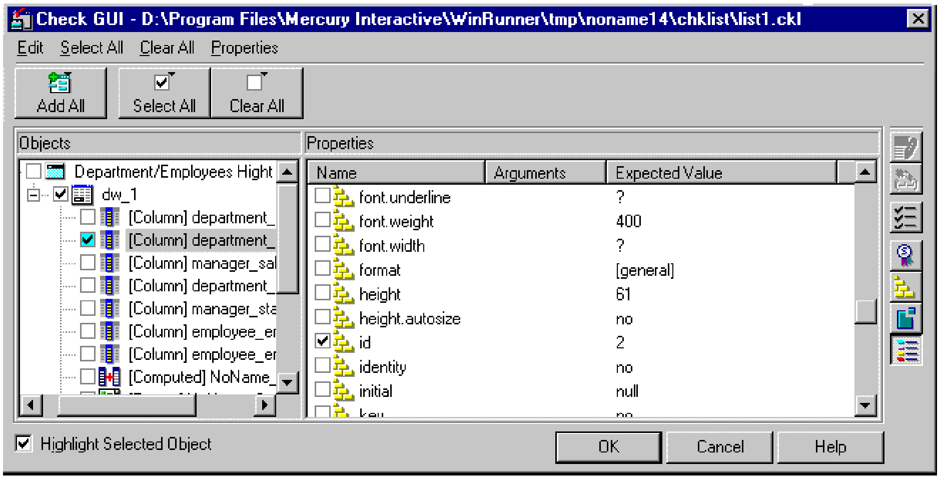
To check the properties of objects in a DataWindow:

1 Choose **Insert** > **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.


Check GUI - D:\Program Files\Mercury Interactive\WinRunner\tmp\noname14\chklist\list1.ckl
Edit Select All Clear All Properties
Add All
Select All
Clear All
Objects
Properties
OK
Cancel
Help

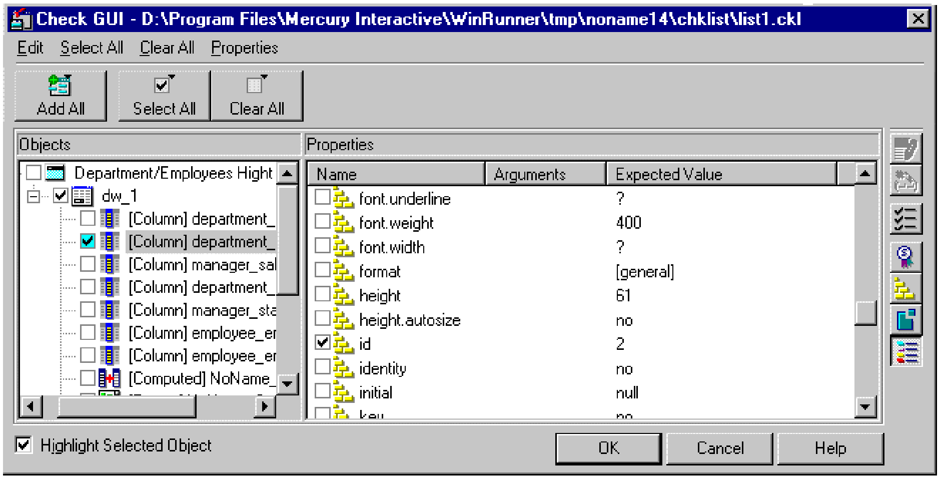
Objects:

- □ Department/Employees Hight
- ☑ dw_1
- □ [Column] department_
- ☑ [Column] department_
- □ [Column] manager_sal
- □ [Column] department_
- □ [Column] manager_sta
- □ [Column] employee_er
- □ [Column] employee_er
- □ [Computed] NoName_

Name	Arguments	Expected Value
□ font.underline		?
□ font.weight		400
□ font.width		?
□ format		[general]
□ height		61
□ height.autosize		no
☑ id		2
□ identity		no
□ initial		null
□ key		no

☑ Highlight Selected Object

3 In the **Objects** pane, click the Expand sign (+) beside the DataWindow to display its objects, and select an object to display its properties.


Check GUI - D:\Program Files\Mercury Interactive\WinRunner\tmp\noname14\chklist\list1.ckl
Edit Select All Clear All Properties
Add All
Select All
Clear All
Objects
Properties
OK
Cancel
Help

Objects:

- □ Department/Employees Hight
- ☑ dw_1
- □ [Column] department_
- ☑ [Column] department_
- □ [Column] manager_sal
- □ [Column] department_
- □ [Column] manager_sta
- □ [Column] employee_er
- □ [Column] employee_er
- □ [Computed] NoName_

Name	Arguments	Expected Value
□ font.underline		?
□ font.weight		400
□ font.width		?
□ format		[general]
□ height		61
□ height.autosize		no
☑ id		2
□ identity		no
□ initial		null
□ key		no

☑ Highlight Selected Object

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 12, "Checking GUI Objects," or refer to the *TSL Reference*.

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.

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 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 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.

16

Checking Table Contents

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:

➤ About Checking Table Contents

➤ 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 41, "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.

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 12, "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 34, "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, "Understanding How WinRunner Identifies GUI Objects," 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 15, "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 14, "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 12, "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 302.

To check table contents with a default check:

1. Choose **Insert** > **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 Reference*.

Note: If you wish to check other table object properties while performing a check on the table contents, use the **Insert** > **GUI Checkpoint** > **For Multiple Objects** command (instead of the **Insert** > **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 12, “Checking GUI Objects.” For information on checking the ActiveX control properties of a tables, see Chapter 14, “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.

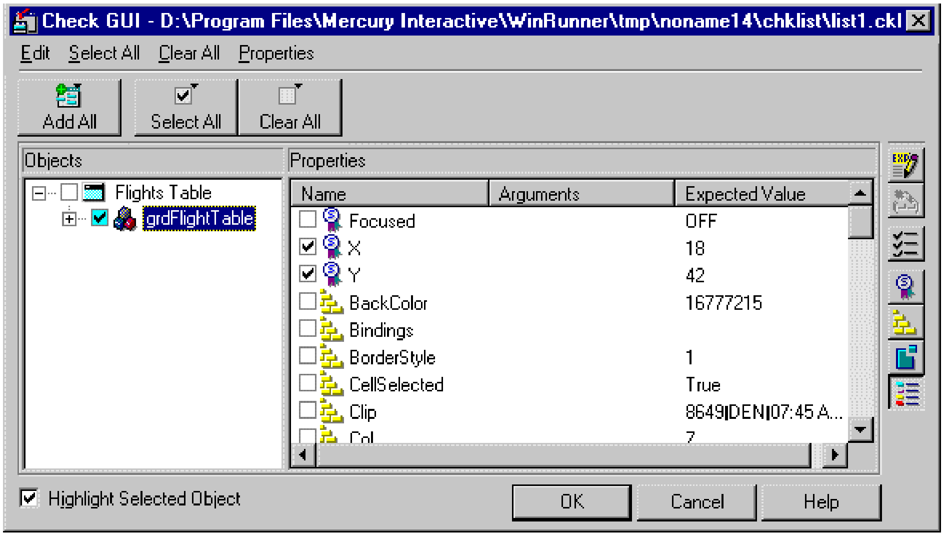
The example in the procedure below uses WinRunner with add-in support for Visual Basic and the Flights table in the sample Visual Basic Flights application.

To check table contents while specifying which checks to perform:

1 Choose **Insert** > **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.

WinRunner may take a few seconds to capture information about the table, and then the Check GUI dialog box opens.



Check GUI - D:\Program Files\Mercury Interactive\WinRunner\tmp\noname14\chklist\list1.ckl

Objects

- □ Flights Table
  - ☑ grdFlightTable

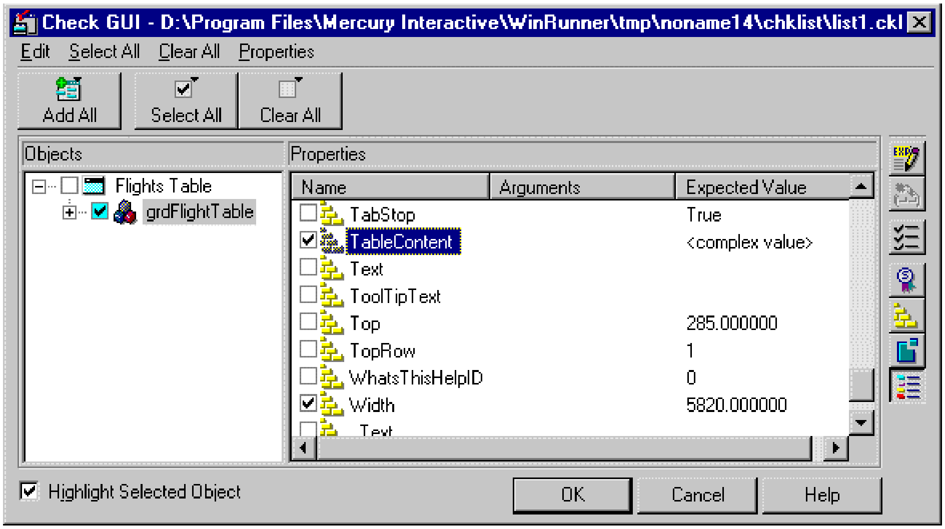
Properties

	Name	Arguments	Expected Value
□	Focused		OFF
☑	X		18
☑	Y		42
□	BackColor		16777215
□	Bindings		
□	BorderStyle		1
□	CellSelected		True
□	Clip		8649|DEN|07:45 A...
□	Col		7

☑ Highlight Selected Object

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.



Check GUI - D:\Program Files\Mercury Interactive\WinRunner\tmp\noname14\chklist\list1.ckl

⊟ □ Flights Table
⊞ ☑ grdFlightTable

Name	Arguments	Expected Value
□ TabStop		True
☑ TableContent		<complex value>
□ Text		
□ ToolTipText		
□ Top		285.000000
□ TopRow		1
□ WhatsThisHelpID		0
☑ Width		5820.000000
□ Text		

☑ Highlight Selected Object

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 304.

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.

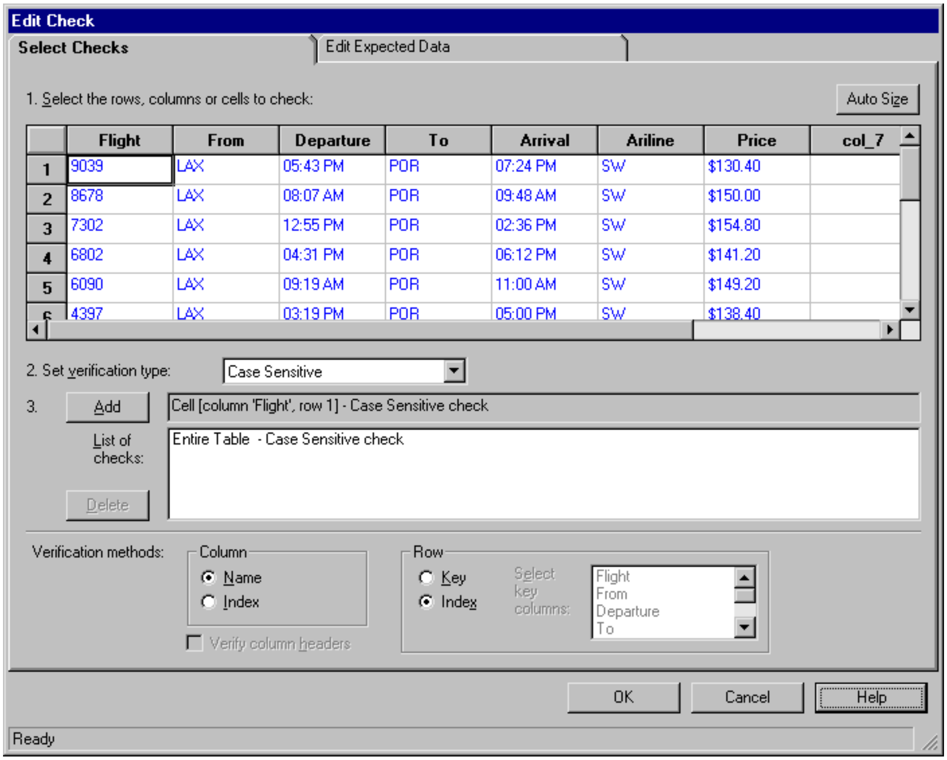
An **obj_check_gui** statement is inserted into your test script. For more information on the **obj_check_gui** function, refer to the *TSL Reference.*

Note: If you wish to check other table object properties while performing a check on the table contents, use the **Insert** > **GUI Checkpoint** > **For Multiple Objects** command (instead of the **Insert** > **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 12, "Checking GUI Objects." For information on checking the ActiveX control properties of a tables, see Chapter 14, "Working with ActiveX and Visual Basic Controls."

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.

(For information on how to open the Edit Check dialog box, see “Checking Table Contents while Specifying Checks” on page 302.)



1. Select the rows, columns or cells to check:

	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. Cell [column 'Flight', row 1] - Case Sensitive check

List of checks: Entire Table - Case Sensitive check

Verification methods:

Column: ● Name ○ Index

□ Verify column headers

Row: ○ Key ● Index

Select key columns: Flight, From, Departure, To

Ready

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 308.

Specifying which Cells to Check

The **List of checks** box 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**.

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 309.

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 306. Choosing this option enables the **Verify column headers** check box, which enables you to check column headers as well as cells.

Row

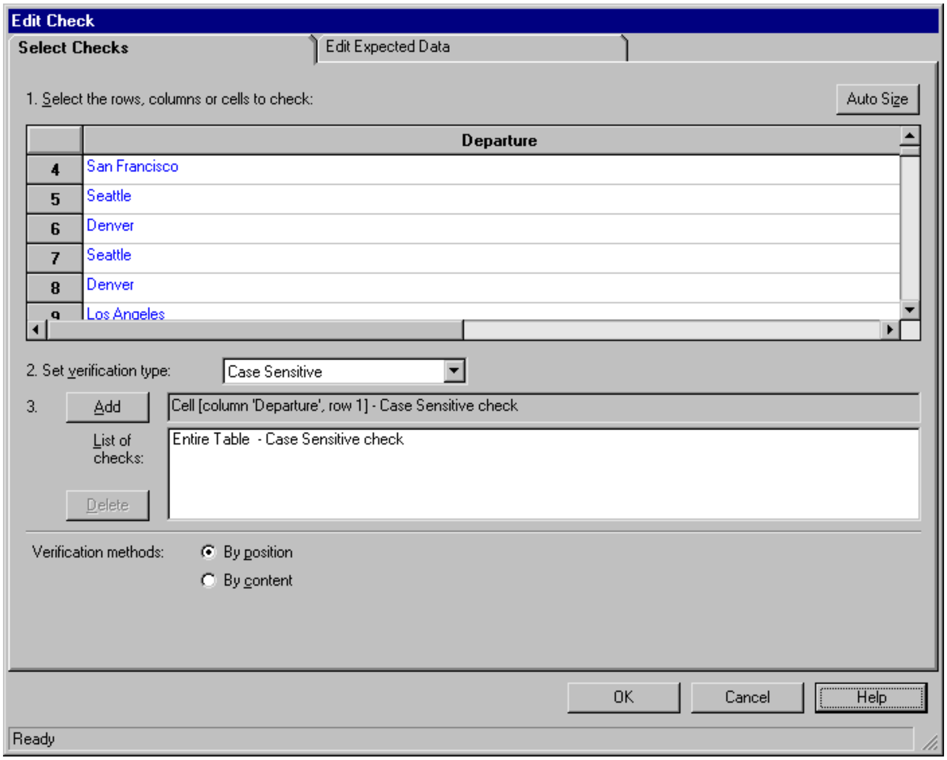
- **Key:** WinRunner looks for the rows in the selection according to the data in the key column(s) specified in the **Select key columns** list box. For example, you could tell WinRunner to identify the second row in the table on page 310 based on the arrival time for that row. A shift in the position of the rows does not result in a mismatch. If the key selection does not uniquely identify a row, WinRunner checks the first matching row. You can use more than one key column to uniquely identify the row.

Note: If the value of a cell in one or more of the key columns changes, WinRunner will not be able to identify the corresponding row, and a check of that row will fail with a “Not Found” error. If this occurs, select a different key column or use the Index verification method.

- **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 methods 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.



	Departure
4	San Francisco
5	Seattle
6	Denver
7	Seattle
8	Denver
9	Los Angeles

2. Set verification type: Case Sensitive

3. Cell [column 'Departure', row 1] - Case Sensitive check

Entire Table - Case Sensitive check

Verification methods: ● By position ○ By content

- **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.

Note: This option causes a mismatch on any string that does not begin with a number. A string starting with 'e' is translated into a number.

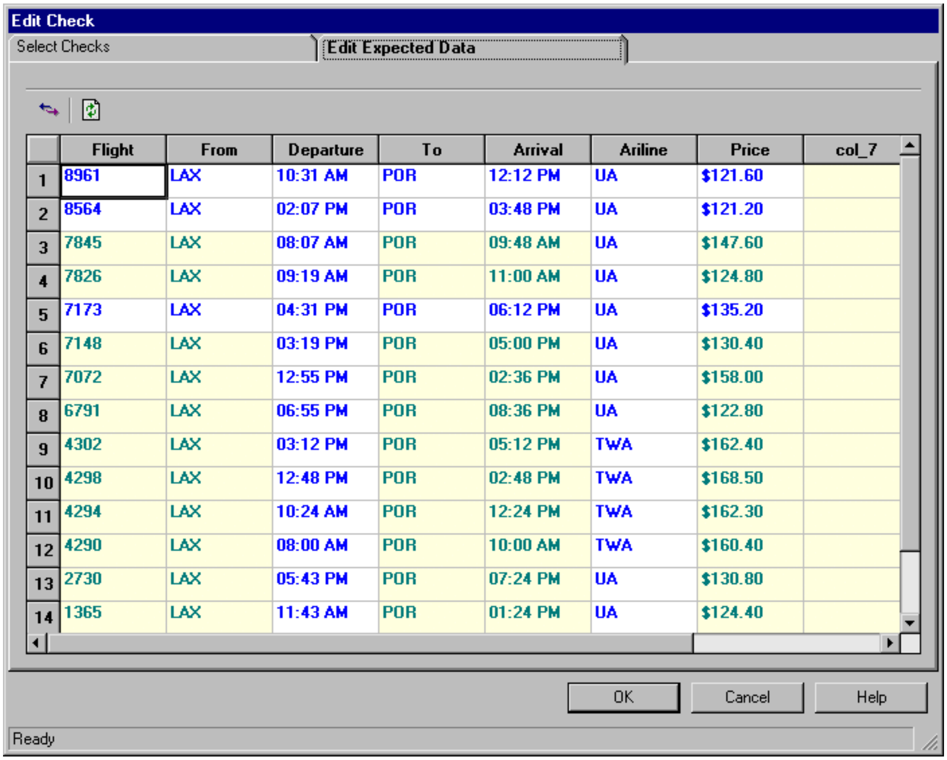
- **Case Sensitive Ignore Spaces:** WinRunner checks the data in the cell according to case and content, ignoring differences in spaces. WinRunner reports any differences in case or content as a mismatch.
- **Case Insensitive Ignore Spaces:** WinRunner checks the content in the cell according to content,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



	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	

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.

17

Checking Databases

By adding runtime database record checkpoints you can compare the information in your application during a test run with the corresponding record in your database.

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

This chapter describes:

About Checking Databases

When you create database checkpoints, you define a query on your database, and your database checkpoint checks the values contained in the *result set*. The result set is a set of values retrieved from the results of the query.

There are several ways to define the query that will be used in your database checkpoints:

- 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. (For standard database checkpoints only). 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.

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.

About Runtime Database Record Checkpoints

You can create runtime database record checkpoints in order to compare the values displayed in your application during the test run with the corresponding values in the database. If the comparison does not meet the success criteria you specify for the checkpoint, the checkpoint fails. You can define a successful runtime database record checkpoint as one where one or more matching records were found, exactly one matching record was found, or where no matching records are found. Your can include your database checkpoint in a loop. If you run your database checkpoint in a loop, the results for each iteration of the checkpoint are displayed in the test results as separate entries. The results of the checkpoint can be viewed in the Test Results window. For more information, see Chapter 34, "Analyzing Test Results."

Runtime record checkpoints are useful when the information in the database changes from one run to the other. Runtime record checkpoints enable you to verify that the information displayed in your application was correctly inserted to the database or conversely, that information from the database is successfully retrieved and displayed on the screen.

Note that when you create a runtime database record checkpoint, the data in the application and in the database are generally in the same format. If the data is in different formats, you can follow the instructions in "Comparing Data in Different Formats" on page 322 to create a runtime database record checkpoint. Note that this feature is for advanced WinRunner users only.

About Standard Database Checkpoints

You can create standard database checkpoints to compare the current values of the properties of the result set during the test run to the expected values captured during recording or otherwise set before the test run. If the expected results and the current results do not match, the database checkpoint fails.

Standard database checkpoints are useful when the expected results can be established before the test run. There are two types of standard database checkpoints: Default and Custom.

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.

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 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. Your can include your database checkpoint in a loop. If you run your database checkpoint in a loop, the results for each iteration of the checkpoint are displayed in the test results as separate entries. The results of the checkpoint can be viewed in the Test Results window. For more information, see Chapter 34, "Analyzing Test Results."

You can modify the expected results of an existing standard 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.

Setting Options for Failed Database Checkpoints

You can instruct WinRunner to send an e-mail to selected recipients each time a database checkpoint fails and you can instruct WinRunner to capture a bitmap of your window or screen when any checkpoint fails. You set these options in the General Options dialog box.

To instruct WinRunner to send an e-mail message when a database checkpoint fails:

1. Choose **Tools** > **General Options**. The General Options dialog box opens.
2. Click the **Notifications** category in the options pane. The notification options are displayed.

3 Select **Database checkpoint failure**.

4 Click the **Notifications** > **E-mail** category in the options pane. The e-mail options are displayed.

5 Select the **Active E-mail service** option and set the relevant server and sender information.

6 Click the **Notifications** > **Recipient** category in the options pane. The e-mail recipient options are displayed.

7 Add, remove, or modify recipient details as necessary to set the recipients to whom you want to send an e-mail message when a database checkpoint fails.

The e-mail contains summary details about the test and checkpoint and details about the connection string and SQL query used for the checkpoint.

For more information, see “Setting Notification Options” on page 808.

To instruct WinRunner to capture a bitmap when a checkpoint fails:

1 Choose **Tools** > **General Options**. The General Options dialog box opens.

2 Click the **Run** > **Settings** category in the options pane. The run settings options are displayed.

3 Select **Capture bitmap on verification failure**.

4 Select **Window**, **Desktop**, or **Desktop area** to indicate what you want to capture when checkpoints fail.

5 If you select **Desktop area**, specify the coordinates of the area of the desktop that you want to capture.

When you run your test, the captured bitmaps are saved in your test results folder.

For more information, see “Setting Test Run Options” on page 793.

Creating a Runtime Database Record Checkpoint

You can add a runtime database record checkpoint to your test in order to compare information displayed in your application during a test run with the current value(s) in the corresponding record(s) in your database.

You add runtime database record checkpoints by running the Runtime Record Checkpoint wizard. When you are finished, the wizard inserts the appropriate **db_record_check** statement into your script.

Note that when you create a runtime database record checkpoint, the data in the application and in the database are generally in the same format. If the data is in different formats, you can follow the instructions in "Comparing Data in Different Formats" on page 322 to create a runtime database record checkpoint. Note that this feature is for advanced WinRunner users only.

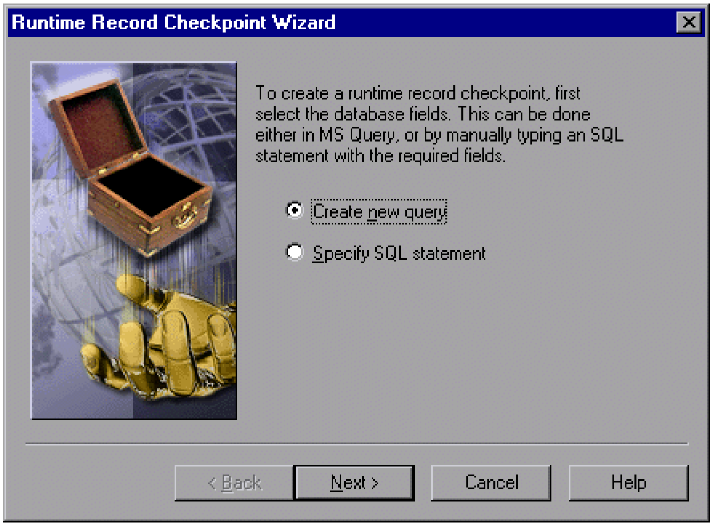
Using the Runtime Record Checkpoint Wizard

The Runtime Record Checkpoint wizard guides you through the steps of defining your query, identifying the application controls that contain the information corresponding to the records in your query, and defining the success criteria for your checkpoint.

To open the wizard, select **Insert** > **Database Checkpoint** > **Runtime Record Check**.

Define Query Screen

The Define Query screen enables you to select a database and define a query for your checkpoint. You can create a new query from your database using Microsoft Query, or manually define an SQL statement.



Runtime Record Checkpoint Wizard

To create a runtime record checkpoint, first select the database fields. This can be done either in MS Query, or by manually typing an SQL statement with the required fields.

● Create new query

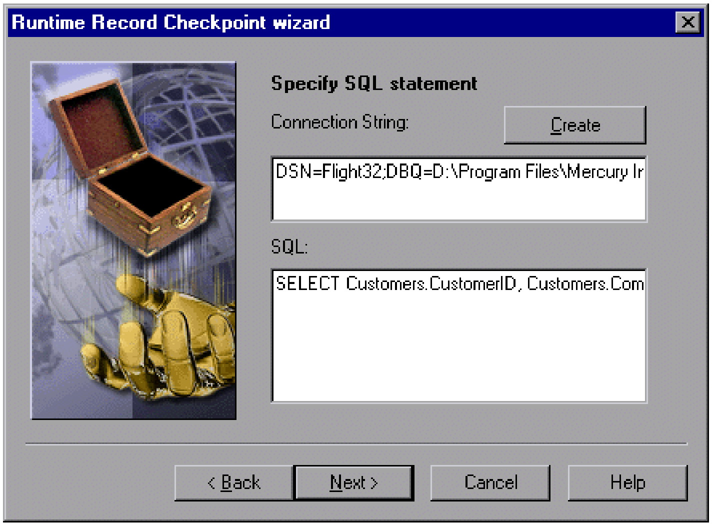
○ Specify SQL statement

You can choose from the following options:

- **Create new query:** Opens Microsoft Query, enabling you to create a new query. Once you finish defining your query, you return to WinRunner. For additional information, see “Creating a Query in ODBC/Microsoft Query,” on page 367. Note that this option is enabled only if Microsoft Query is installed on your machine.
- **Specify SQL statement:** Opens the Specify SQL Statement screen in the wizard, enabling you to specify the connection string and an SQL statement. For additional information, see “Specifying an SQL Statement” on page 340.

Specify SQL Statement Screen

The Specify SQL Statement screen enables you to manually specify the database connection string and the SQL statement.



Runtime Record Checkpoint wizard

Specify SQL statement

Connection String:

DSN=Flight32;DBQ=D:\Program Files\Mercury Ir

SQL:

SELECT Customers.CustomerID, Customers.Com

Enter the required information:

- **Connection String:** Enter the connection string, or click the **Create** button
  - **Create:** Opens the ODBC Select Data Source dialog box. You can select a *.dsn file in the Select Data Source dialog box to have it insert the connection string in the box for you.
- **SQL:** Enter the SQL statement.

Note: You cannot use an SQL statement of the type "SELECT * from ..." with the **db_record_check** function. Instead, you must supply the tables and field names. The reason for this is that WinRunner needs to know which database fields should be matched to which variables in the WinRunner script. The expected SQL format is: SELECT table_name1.field_name1, table_name2.field_name2, ... FROM table_name1, table_name2, ... [WHERE ...]

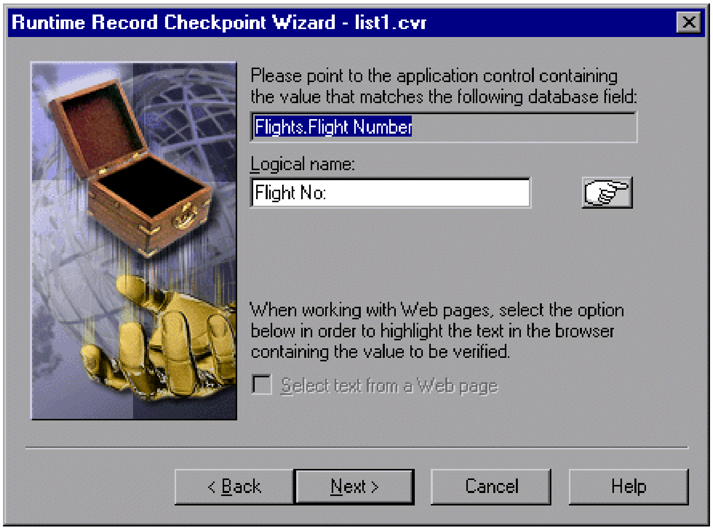
Match Database Field Screen

The Match Database Field screen enables you to identify the application control or text in your application that matches the displayed database field. You repeat this step for each field included in your query.

This screen includes the following options:

- **Database field:** Displays a database field from your query. Use the pointing hand to identify the control or text that matches the displayed field name.
- **Logical name**: Displays the logical name of the control you select on your application.

  (Displayed only when the **Select text from a Web page** check box is cleared.)



Runtime Record Checkpoint Wizard - list1.cvr

Please point to the application control containing the value that matches the following database field:

Flights.Flight Number

Logical name:

Flight No:

When working with Web pages, select the option below in order to highlight the text in the browser containing the value to be verified.

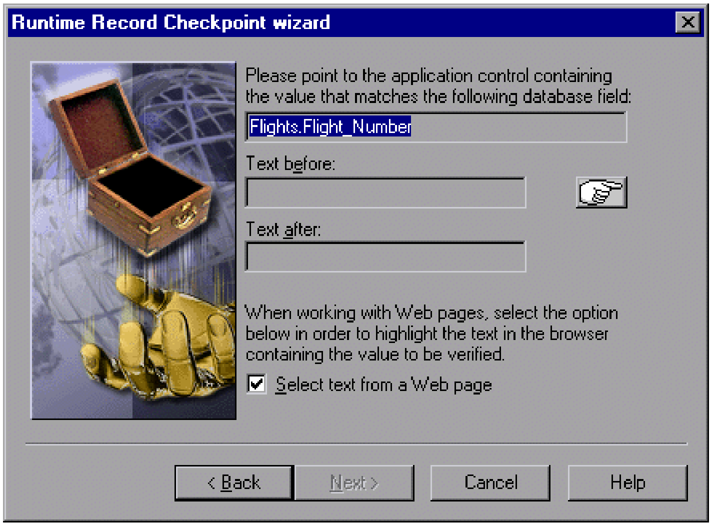
□ Select text from a Web page

- **Text before**: Displays the text that appears immediately before the text to check.

  (Displayed only when the Select text from a Web page check box is checked.)

- **Text after**: Displays the text that appears immediately after the text to check.

  (Displayed only when the **Select text from a Web page** check box is selected.)



Runtime Record Checkpoint wizard

Please point to the application control containing the value that matches the following database field:

Flights.Flight_Number

Text before:

Text after:

When working with Web pages, select the option below in order to highlight the text in the browser containing the value to be verified.

☑ Select text from a Web page

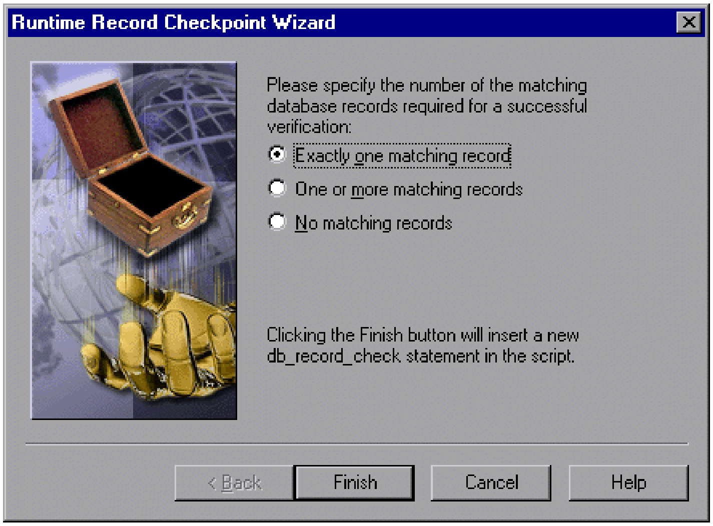
- **Select text from a Web page:** Enables you to indicate the text on your Web page containing the value to be verified.

Note: When selecting text from a Web page, you must use the pointer to select the text.

Note: To create a database checkpoint on a database field mapped to a text string in a Web page, the WebTest Add-in must be loaded. If necessary, you must restart WinRunner with the WebTest Add-in loaded before creating the checkpoint. For information on loading add-ins, see "Loading WinRunner Add-Ins" on page 20.

Matching Record Criteria Screen

The Matching Record Criteria screen enables you to specify the number of matching database records required for a successful checkpoint.



Runtime Record Checkpoint Wizard

Please specify the number of the matching database records required for a successful verification:

● Exactly one matching record

○ One or more matching records

○ No matching records

Clicking the Finish button will insert a new db_record_check statement in the script.

- **Exactly one matching record:** Sets the checkpoint to succeed if exactly one matching database record is found.
- **One or more matching records:** Sets the checkpoint to succeed if one or more matching database records are found.
- **No matching records:** Sets the checkpoint to succeed if no matching database records are found.

When you click **Finish** on the Runtime Record Checkpoint wizard, a **db_record_check** statement is inserted into your script. For more information on the **db_record_check** function, refer to the *TSL Reference* or the *TSL Reference Guide*.

Comparing Data in Different Formats

Suppose you want to compare the data in your application to data in the database, but the data is in different formats. You can follow the instructions below to create a runtime database record checkpoint without using the Runtime Record Checkpoint Wizard. Note that this feature is for advanced WinRunner users only.

For example, in the sample Flight Reservation application, there are three radio buttons in the Class box. When this box is enabled, one of the radio buttons is always selected. In the database of the sample Flight Reservation application, there is one field with the values 1, 2, or 3 for the matching class.

To check that data in the application and the database have the same value, you must perform the following steps:

1. Record on your application up to the point where you want to verify the data on the screen. Stop your test. In your test, manually extract the values from your application.
2. Based on the values extracted from your application, calculate the expected values for the database. Note that in order to perform this step, you must know the mapping relationship between both sets of values. See the example below.
3. Add these calculated values to any edit field or editor (e.g. Notepad). You need to have one edit field for each calculated value. For example, you can use multiple Notepad windows, or another application that has multiple edit fields.
4. Use the GUI Map Editor to teach WinRunner:
   - the controls in your application that contain the values to check
   - the edit fields that will be used for the calculated values
5. Add TSL statements to your test script to perform the following operations:
   - extract the values from your application
   - calculate the expected database values based on the values extracted from your application
   - write these expected values to the edit fields

6 Use the Runtime Record Checkpoint wizard, described in “Using the Runtime Record Checkpoint Wizard,” on page 316, to create a **db_record_check** statement.

When prompted, instead of pointing to your application control with the desired value, point to the edit field where you entered the desired calculated value.

Tip: When you run your test, make sure to open the application(s) with the edit field(s) containing the calculated values.

Example of Comparing Different Data Formats in a Runtime Database Record Checkpoint

The following excerpts from a script are used to check the Class field in the database against the radio buttons in the sample Flights application. The steps refer to the instructions on page 322.

Step 1

```
# Extract values from GUI objects in application.
button_get_state("First",vFirst);
button_get_state("Business",vBusiness);
button_get_state("Economy",vEconomy);
```

Step 2

```
# Calculate the expected values for the database.
if (vFirst)
    expDBval = "1" ;
else if (vBusiness)
    expDBval = "2" ;
else if (vEconomy)
    expDBval = "3" ;
```

Step 3

```
# Add these calculated values to an edit field to be used in the checkpoint.
set_window("Untitled - Notepad", 1);
edit_set("Edit", expDBval);
```

Step 4

```
# Create a runtime database record checkpoint using the wizard.
db_record_check("list1.cvr", DVR_ONE_MATCH);
```

Editing a Runtime Database Record Checklist

You can make changes to a checklist you created for a runtime database record checkpoint. Note that a checklist includes the connection string to the database, the SQL statement or a query, the database fields in the data source, the controls in your application, and the mapping between them. It does not include the success conditions of a runtime database record checkpoint.

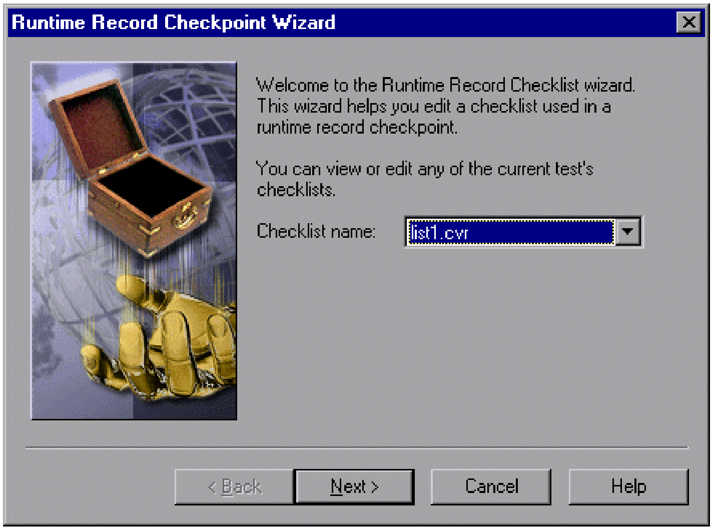
When you edit a runtime database record checklist, you can:

- modify the data source connection string manually or using ODBC
- modify the SQL statement or choose a different query in Microsoft Query
- select different database fields to use in the data source (add or remove)
- match a database field already in the checklist to a different application control
- match a new database field in the checklist to an application control

To edit an existing runtime database record checklist:

1 Choose **Insert** > **Edit Runtime Record Checklist**.

The Runtime Record Checkpoint wizard opens.



Welcome to the Runtime Record Checklist wizard. This wizard helps you edit a checklist used in a runtime record checkpoint.

You can view or edit any of the current test's checklists.

Checklist name: list1.cvr

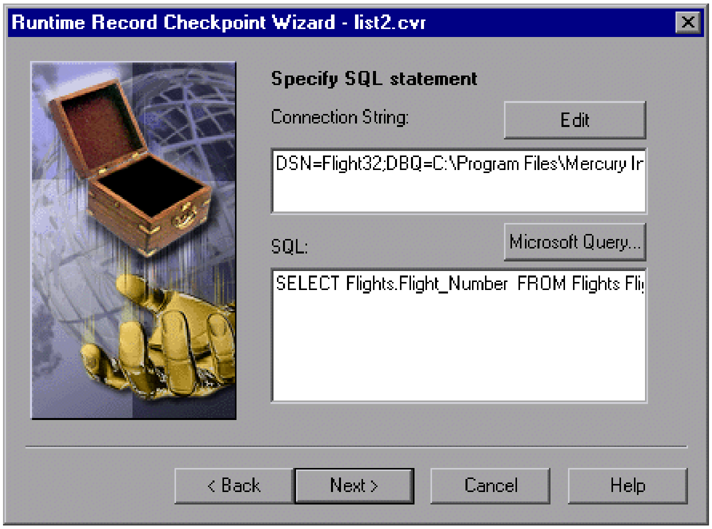
2 Choose the runtime database record checklist to edit.

Note: By default, runtime database record checklists are named sequentially in each test, starting with *list1.cvr*.

Tip: You can see the name of the checklist you want to edit in the **db_record_check** statement in your test script.

Click **Next** to proceed.

3 The Specify SQL Statement screen opens:



Runtime Record Checkpoint Wizard - list2.cvr

Specify SQL statement

Connection String:

DSN=Flight32;DBQ=C:\Program Files\Mercury Ir

SQL:

SELECT Flights.Flight_Number FROM Flights Fli

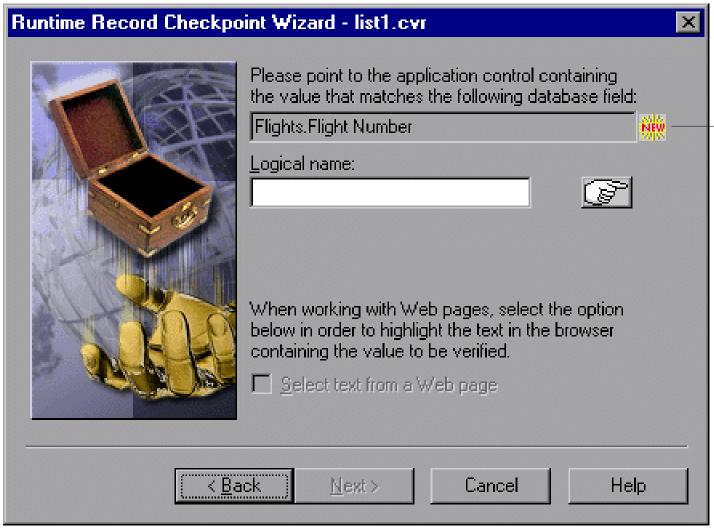
In this screen you can:

- modify the connection string manually or by clicking **Edit** to open the ODBC Select Data Source dialog box, where you can select a new *.dsn file in the Select Data Source dialog box to create a new connection string
- modify the SQL statement manually or redefine the query by clicking the **Microsoft Query** button to open Microsoft Query

Note: If Microsoft Query is not installed on your machine, the **Microsoft Query** button is not displayed.

Click **Next** to continue.

4 The Match Database Field screen opens:



Runtime Record Checkpoint Wizard - list1.cvr

Please point to the application control containing the value that matches the following database field:

Flights.Flight Number

Logical name:

When working with Web pages, select the option below in order to highlight the text in the browser containing the value to be verified.

□ Select text from a Web page

"New" icon indicates that this database field was not previously included in the checklist.

➤ For a database field previously included in the checklist, the database field is displayed along with the application control to which it is mapped. You can use the pointing hand to map the displayed field name to a different application control or text string in a Web page.

Note: To edit a database field mapped to a text string in a Web page, the WebTest Add-in must be loaded. If necessary, you must restart WinRunner with the WebTest Add-in loaded before editing this object in the checklist. For information on loading add-ins, see "Loading WinRunner Add-Ins" on page 20.

NEW

- If you modified the SQL statement or query in Microsoft Query so that it now references an additional database field in the data source, the checklist will now include a new database field. You must match this database field to an application control. Use the pointing hand to identify the control or text that matches the displayed field name.

Tip: New database fields are marked with a "New" icon.

Note: To map the database field to text in a Web page, click the **Select text from a Web page** check box, which is enabled when you load the WebTest Add-in. The wizard screen will display additional options. For information on these options, see "Match Database Field Screen" on page 319.

Click **Next** to continue.

Note: The Match Database Field screen is displayed once for each database field in the SQL statement or query in Microsoft Query. Follow the instructions in this step each time this screen is displayed.

5 The Finished screen is displayed.

Click **Finish** to modify the checklist used in the runtime record checkpoint(s).

Note: You can change the success condition of your checkpoint by modifying the second parameter in the **db_record_check** statement in your test script. The second parameter must contain one of the following values:

- DVR_ONE_OR_MORE_MATCH - The checkpoint passes if one or more matching database records are found.
- DVR_ONE_MATCH - The checkpoint passes if exactly one matching database record is found.
- DVR_NO_MATCH - The checkpoint passes if no matching database records are found.

For additional information, refer to the *TSL Reference*.

Tip: You can use an existing checklist in multiple runtime record checkpoints. Suppose you already created a runtime record checkpoint in your test script, and you want to use the same data source and SQL statement or query in additional runtime record checkpoints in the same test. For example, suppose you want several different **db_record_check** statements, each with different success conditions. You do not need to rerun the Runtime Record Checkpoint wizard for each new checkpoint you create. Instead, you can manually enter a **db_record_check** statement that references an existing checklist. Similarly, you can modify an existing **db_record_check** statement to reference an existing checklist.

Creating a Default Check on a Database

When you create a default check on a database, you create a standard database checkpoint that checks the entire result set using the following criteria:

- 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 333.

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 **Insert** > **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 336.

---

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.

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 "Setting ODBC (Microsoft Query) Options" on page 337.

3 Define a query, copy a query, or specify an SQL statement. For additional information, see "Creating a Query in ODBC/Microsoft Query" on page 367 or "Specifying an SQL Statement" on page 340.

Note: If you want to be able to parameterize the SQL statement in the **db_check** statement that is 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 366.

---

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 *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 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 **Insert** > **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 336.

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 "Creating a Conversion File in Data Junction" on page 369.

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 Reference*.

Creating a Custom Check on a Database

When you create a custom check on a database, you create a standard 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

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

To create a custom check on a database:

1. Choose **Insert** > **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 336.

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.

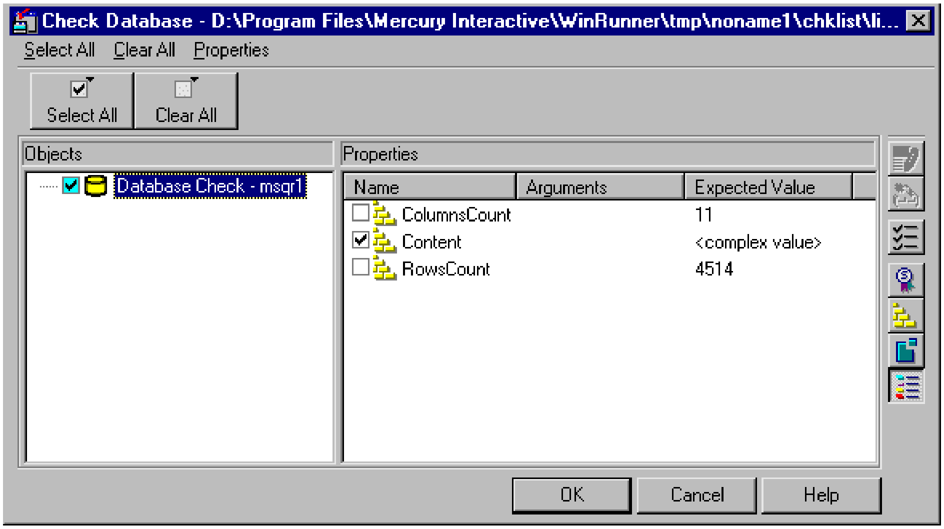
4. If you are using ODBC or Microsoft Query, define a query, copy a query, or specify an SQL statement. For additional information, see “Creating a Query in ODBC/Microsoft Query” on page 367 or “Specifying an SQL Statement” on page 340.

   If you are using Data Junction, create a new conversion file or use an existing one. For additional information, see “Creating a Conversion File in Data Junction” on page 369.

5 If you are using Microsoft Query and 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 Standard Database Checkpoints" on page 363.

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

The Check Database dialog box opens.



Check Database - D:\Program Files\Mercury Interactive\WinRunner\tmp\noname1\chklist\li...
Select All
Clear All
Properties
Select All
Clear All
Objects
Properties
OK
Cancel
Help

☑ Database Check - msqr1

	Name	Arguments	Expected Value
□	ColumnsCount		11
☑	Content		<complex value>
□	RowsCount		4514

The **Objects** pane contains "Database check" and the name of the **.sql* query file or **.djs* conversion file 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.

7 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 330.

RowsCount: Counts the number of rows in the result set.

8 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 343.

9 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 Reference*.

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 34, "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

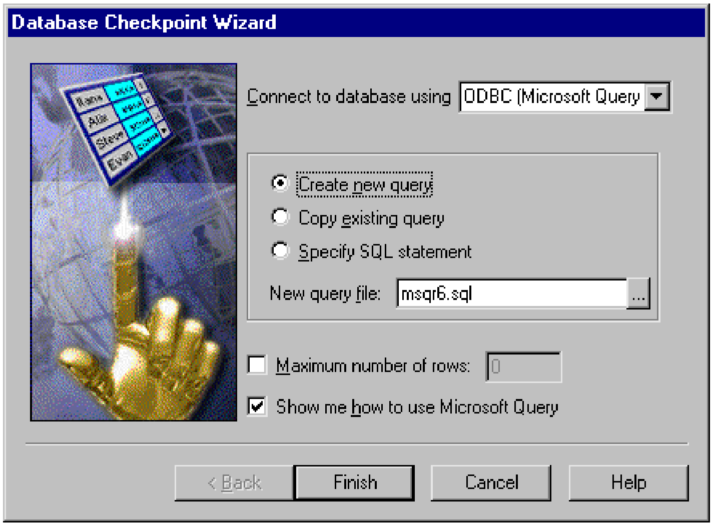
ODBC (Microsoft Query) Screens

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

Setting ODBC (Microsoft Query) Options

The following screen opens if you are creating a custom database checkpoint or working in ODBC mode.



Database Checkpoint Wizard

Connect to database using: ODBC (Microsoft Query

● Create new query
○ Copy existing query
○ Specify SQL statement

New query file: msqr6.sql

□ Maximum number of rows: 0

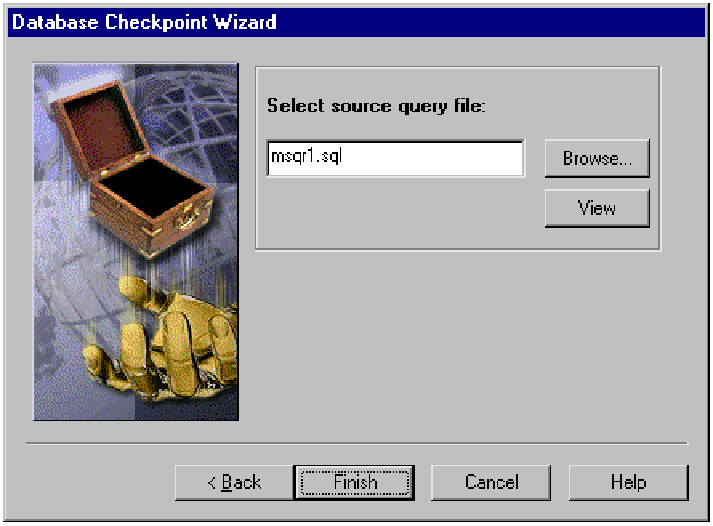
☑ Show me how to use Microsoft Query

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 333.
- **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 339.
- **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 340.
- **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 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.



Select source query file: msqr1.sql

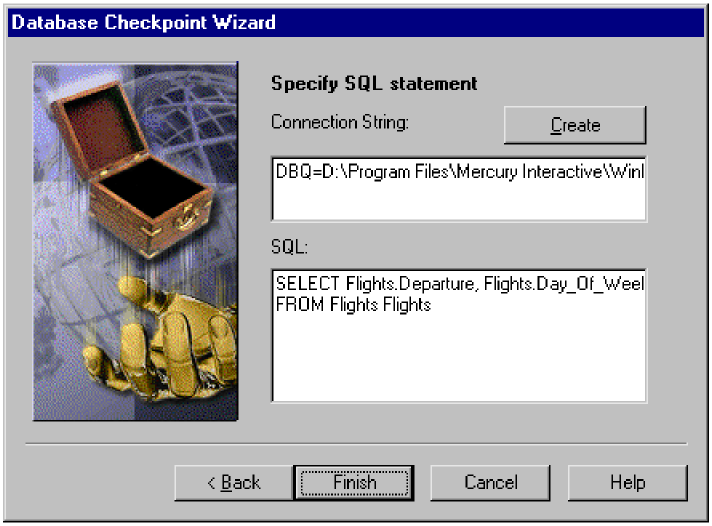
Enter the pathname of the query file or use the **Browse** button to locate it. Once a query file is selected, you can use the **View** button to open the file for viewing.

➤ 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 333.

Specifying an SQL Statement

The following screen opens if you chose to specify an SQL statement to use in this database checkpoint.



Connection String:

DBQ=D:\Program Files\Mercury Interactive\Winl

SQL:

```
SELECT Flights.Departure, Flights.Day_Of_Weel
FROM Flights Flights
```

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 Standard Database Checkpoints" on page 363.

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 333.

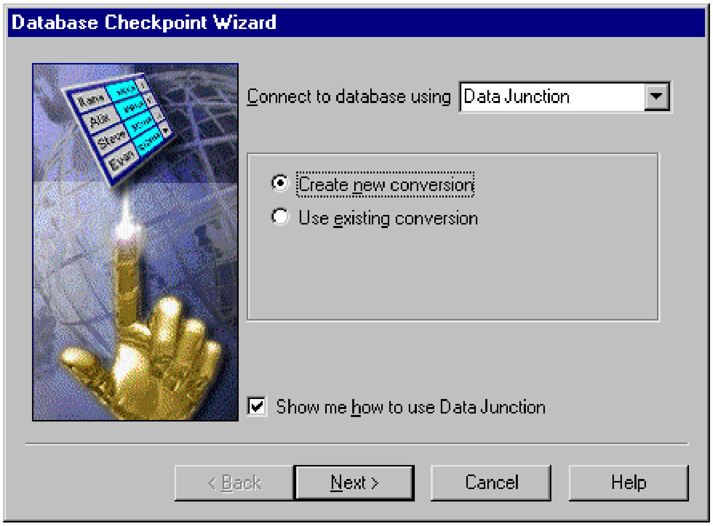
Data Junction Screens in the Database Checkpoint Wizard

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:



Database Checkpoint Wizard

Connect to database using: Data Junction

● Create new conversion
○ Use existing conversion

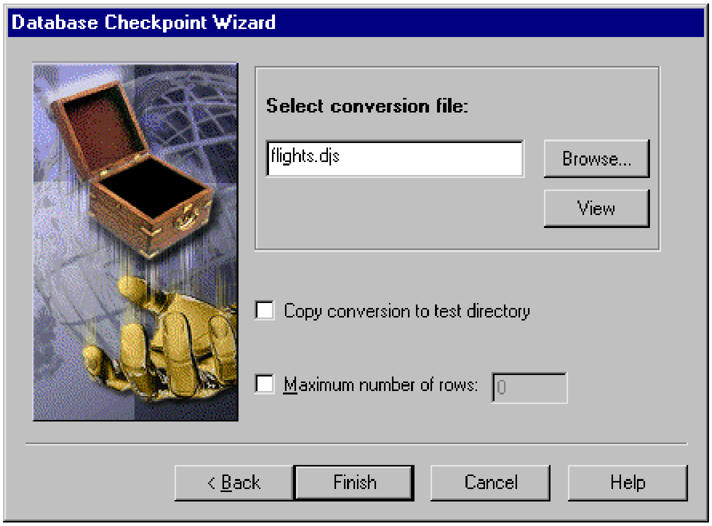
☑ Show me how to use Data Junction

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 369. 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 342.
- **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 342.
- **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.



Select conversion file: flights.djs

□ Copy conversion to test directory

□ Maximum number of rows: 0

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:

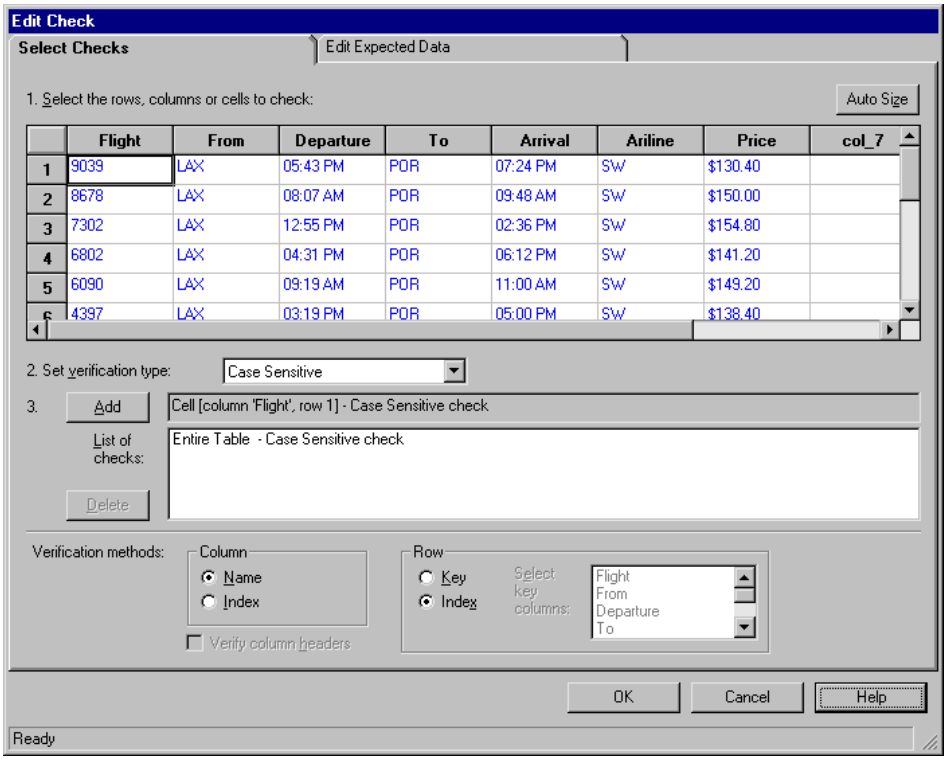
- **Copy conversion to test folder**: Copies the specified conversion file to the test folder.
- **Maximum number of rows:** Select this check box and enter the maximum number of database rows to check. If this check box is cleared, there is no maximum.

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 333.

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. (For information on how to open the Edit Check dialog box, see "Creating a Custom Check on a Database" on page 333.)



1. Select the rows, columns or cells to check:

	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. Cell [column 'Flight', row 1] - Case Sensitive check

List of checks: Entire Table - Case Sensitive check

Verification methods:
Column
● Name
○ Index
□ Verify column headers

Row
○ Key
● Index
Select key columns: Flight From Departure To

Ready

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 347.

Specifying which Cells to Check

The **List of checks** box 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** 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**.

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 348.

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** box.

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

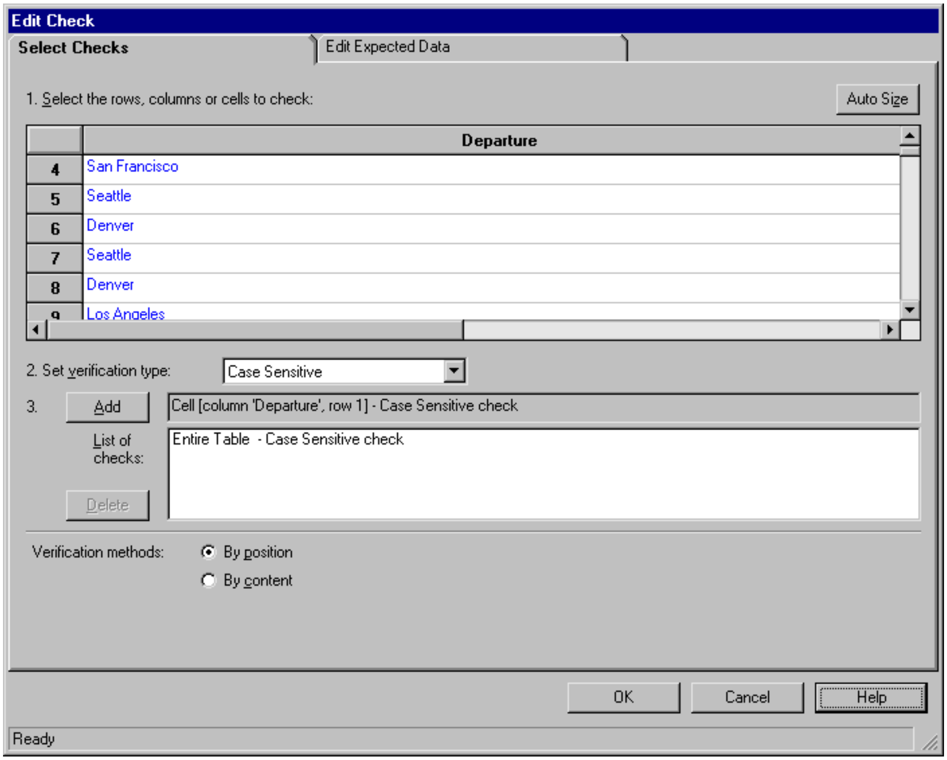
- **Name:** (default setting): WinRunner looks for the selection according to the column names. A shift in the position of the columns within the result set does not result in a mismatch.
- **Index:** WinRunner looks for the selection according to the index, or position, of the columns. A shift in the 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 345. 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 methods 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.



1. Select the rows, columns or cells to check:

	Departure
4	San Francisco
5	Seattle
6	Denver
7	Seattle
8	Denver
9	Los Angeles

2. Set verification type: Case Sensitive

3. Cell [column 'Departure', row 1] - Case Sensitive check

List of checks: Entire Table - Case Sensitive check

Verification methods: ● By position ○ By content

Ready

- **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 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.

Note: This option causes a mismatch on any string that does not begin with a number. A string starting with 'e' is translated into a number.

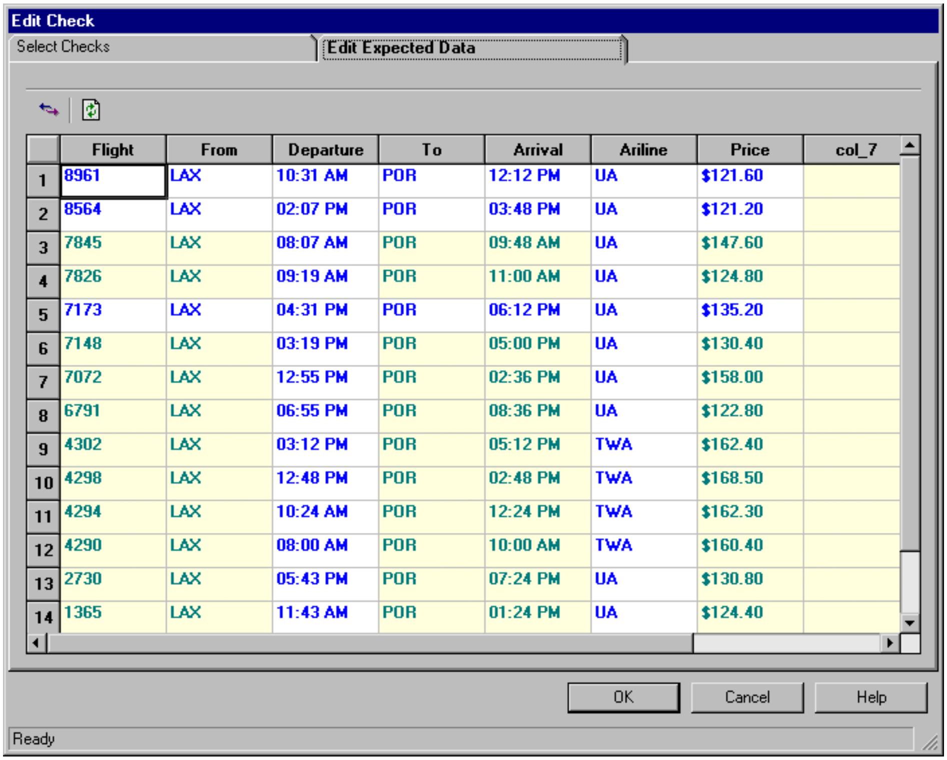
- **Case Sensitive Ignore Spaces:** WinRunner checks the data in the field according to case and content, ignoring differences in spaces. WinRunner reports any differences in case or content as a mismatch.
- **Case Insensitive Ignore Spaces:** WinRunner checks the content in the cell according to content,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

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.



	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	

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 Standard Database Checkpoint

You can make the following changes to an existing standard database checkpoint:

- 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 Standard Database Checkpoint" on page 361.

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 the same checklist in multiple tests.

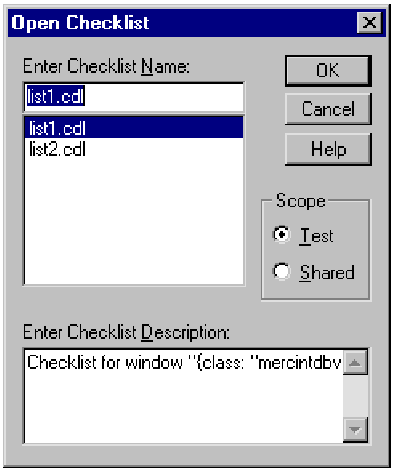
Note: **.sql* files are not saved in shared database checklist folders.

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** category of the General Options dialog box. For more information, see Chapter 41, "Setting Global Testing Options."

To save a database checklist in a shared folder:

1 Choose **Insert** > **Edit Database Checklist**.

The Open Checklist dialog box opens.



Enter Checklist Name: list1.cdl

list1.cdl
list2.cdl

Scope: ● Test ○ Shared

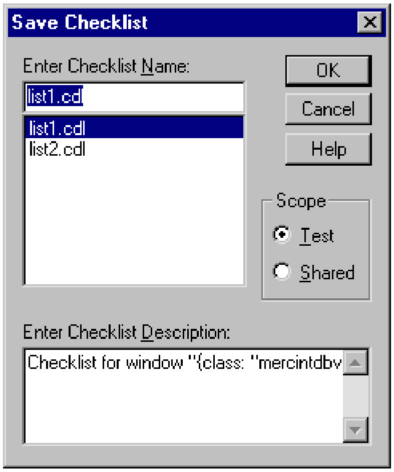
Enter Checklist Description: Checklist for window "{class: "mercintdbv

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 190.

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.



Enter Checklist Name: list1.cd

list1.cdl
list2.cdl

Scope
● Test
○ Shared

Enter Checklist Description:
Checklist for window "{class: "mercintdbv

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.

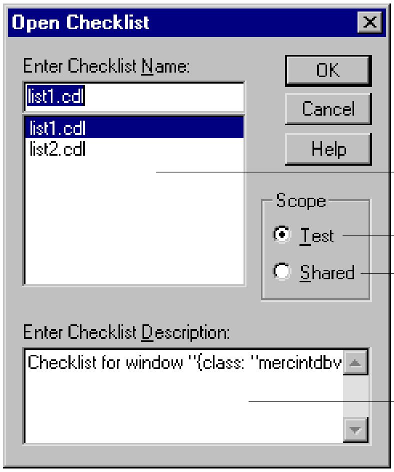
You may want to edit a database checklist to change which properties in a database to check.

To edit an existing database checklist:

1 Choose **Insert > 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 350.



Checklist for window "{class: "mercintdbv

Lists the available checklists.

Displays checklists created for the current test.

Displays checklists created in a shared folder.

Describes the selected checklist.

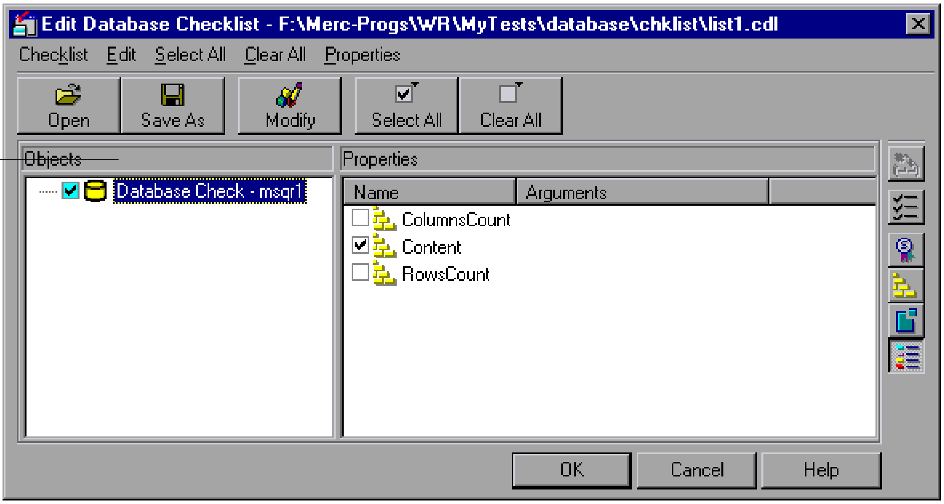
3 Select a database checklist.

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 355.

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 330.

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 621.

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

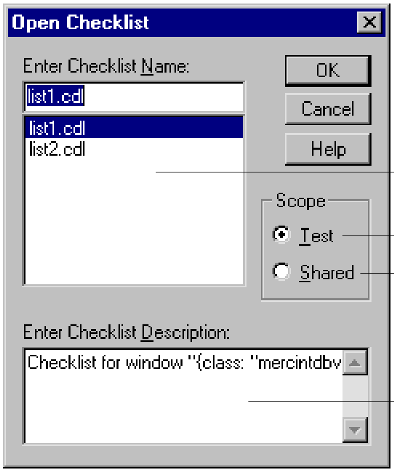
You can modify a query created with ODBC/Microsoft Query from the Edit Database Checklist dialog box.

To modify a database checkpoint created with ODBC/Microsoft Query:

1 Choose **Insert** > **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 350.



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 database checklist.

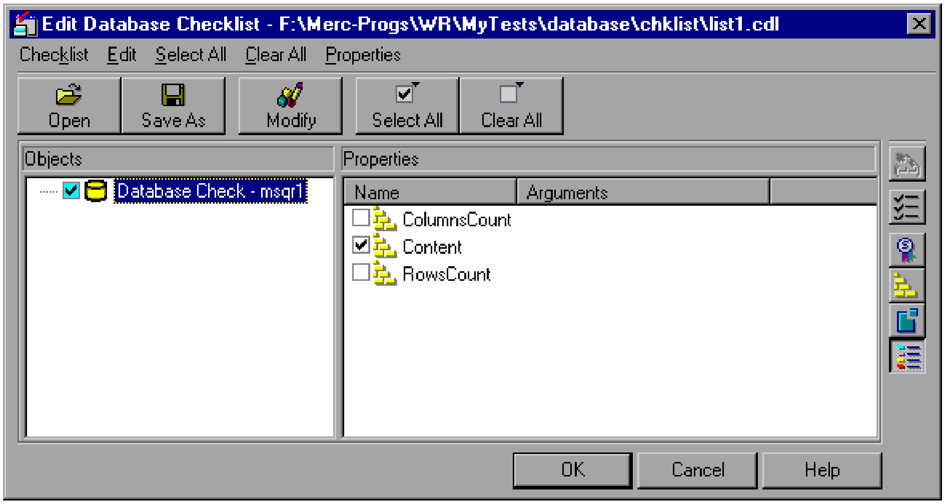
4 Click **OK**.

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 352.

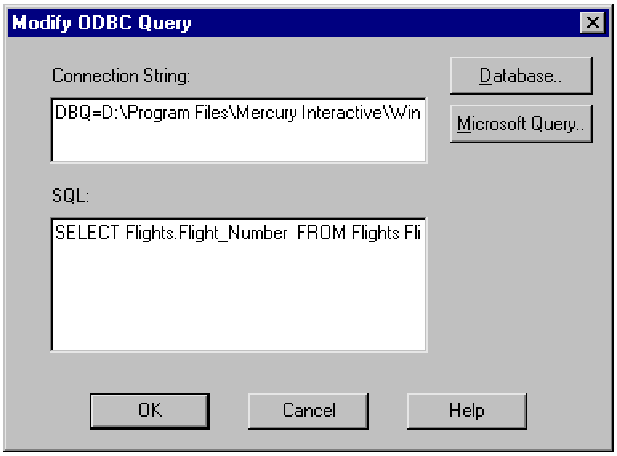

Edit Database Checklist - F:\Merc-Progs\WR\MyTests\database\chklist\list1.cdl
Checklist Edit Select All Clear All Properties
Open
Save As
Modify
Select All
Clear All
Objects
Properties
Name
Arguments
OK
Cancel
Help

☑ Database Check - msqr1

Name	Arguments
□ ColumnsCount	
☑ Content	
□ RowsCount	

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.



Connection String: DBQ=D:\Program Files\Mercury Interactive\Win

SQL: SELECT Flights.Flight_Number FROM Flights Fli

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. For more information, see "Running a Test to Update Expected Results" on page 630.

Modifying a Query Created with Data Junction

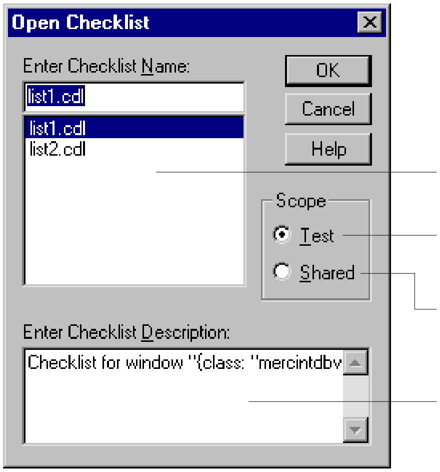
You can modify a Data Junction conversion file used in a database checkpoint directly in Data Junction. To see the pathname of the conversion file, follow the instructions below.

To see the pathname of a Data Junction conversion file in a database checkpoint:

1 Choose **Insert** > **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 350.



Enter Checklist Name: list1.cdl

list1.cdl
list2.cdl

● Test
○ Shared

Enter Checklist Description: Checklist for window "{class: "mercintdbv

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 database checklist.

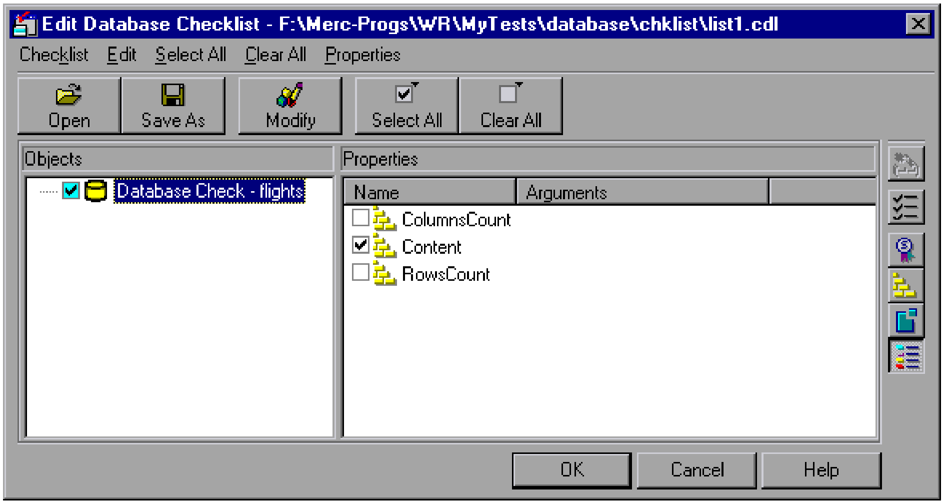
4 Click **OK**.

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 352.



☑ Database Check - flights

□ ColumnsCount
☑ Content
□ RowsCount

Modify

5 In the **Objects** column, highlight the name of the conversion file, and click **Modify**.

WinRunner 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. For more information, see "Running a Test to Update Expected Results" on page 630.

Modifying the Expected Results of a Standard Database Checkpoint

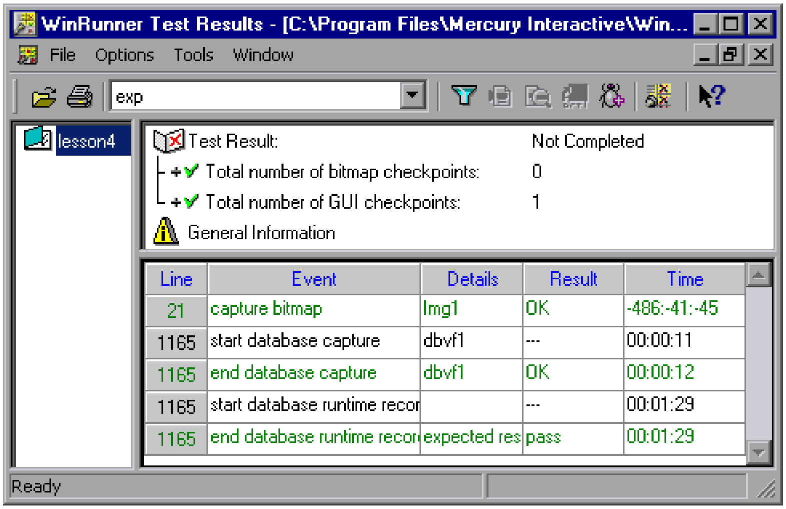
You can modify the expected results of an existing standard database checkpoint by changing the expected value of a property check within the checkpoint. You can make this change before or after you run your test.

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 Select **exp** in the results location box.



WinRunner Test Results - [C:\Program Files\Mercury Interactive\Win...

exp

lesson4

Test Result: Not Completed

Total number of bitmap checkpoints: 0

Total number of GUI checkpoints: 1

General Information

Line	Event	Details	Result	Time
21	capture bitmap	Img1	OK	-486:-41:-45
1165	start database capture	dbvf1	---	00:00:11
1165	end database capture	dbvf1	OK	00:00:12
1165	start database runtime recor		---	00:01:29
1165	end database runtime recor	expected res	pass	00:01:29

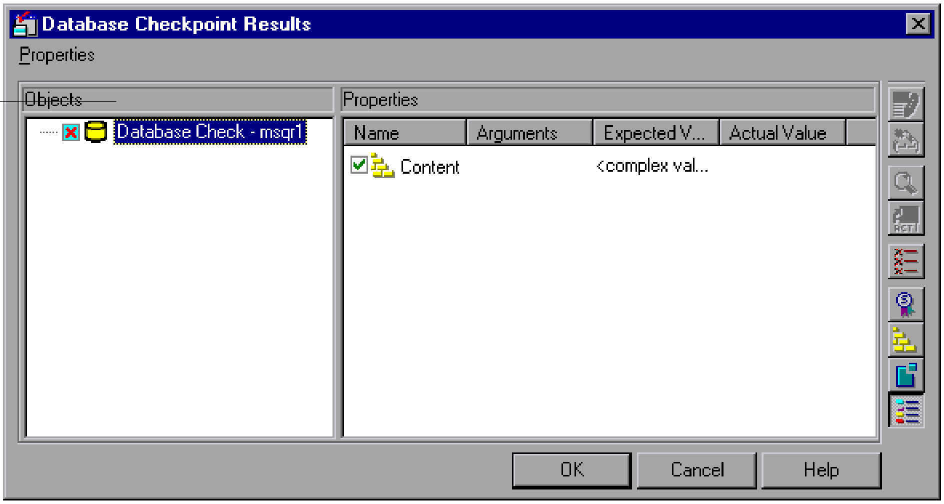
Ready

3 Locate the database checkpoint by looking at **end database capture** entries.

Note: If you are working in the WinRunner report view, you can use the **Show TSL** button to open the test script to the highlighted line number.

---

4 Select and display the **end database capture** entry. 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 333.

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 in the WinRunner Report View" on page 683.

For more information on working in the Test Results window, see Chapter 34, "Analyzing Test Results."

Parameterizing Standard Database Checkpoints

When you create a standard database checkpoint using ODBC (Microsoft Query), you can add parameters to an SQL statement to parameterize the 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 Denver on Monday when you create the query. You might also want to use an identical query to check all the flights departing from San Francisco on Tuesday. Instead of creating a new query or rewriting the SQL statement in the existing query to reflect the changes in day of the week or departure points, 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: “Denver,” or “San Francisco.” Similarly, you can use a parameter for the day of the week value, and replace the parameter with either “Monday” or Tuesday.”

Understanding Parameterized Queries

A parameterized query is a query in which at least one of the fields of the WHERE clause is parameterized, i.e., the value of the field is specified by a question mark symbol (**?**). 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=?) AND (Flights.Day_Of_Week=?)
```

- **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.
- **Departure** is the parameter that represents the departure point of a flight.
- **Day_Of_Week** is the parameter that represents the day of the week of a flight.

In order to execute a parameterized query, you must specify the values for the parameters.

Note for Microsoft Query users: When you use Microsoft Query to create a query, the parameters are specified by brackets. When Microsoft Query generates an SQL statement, the bracket symbols are replaced by a question mark symbol (**?**). 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.

Creating a Parameterized Database Checkpoint

You use a parameterized query to create a parameterized checkpoint. 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, argument: the *parameter_array* argument. A statement similar to the following is inserted into your test script:

```
db_check("list1.cdl", "dbvf1", NO_LIMIT, dbvf1_params);
```

The *parameter_array* argument will contain the values to substitute for the parameters in the parameterized checkpoint.

WinRunner cannot capture the expected result set when you record your test. Unlike regular database checkpoints, recording a parameterized checkpoint requires additional steps to capture the expected results set. Therefore, you must use array statements to add the values to substitute for the parameters. The array statements could be similar to the following:

```
dbvf1_params[1] = “Denver”;
dbvf1_params[2] = “Monday”;
```

You insert the array statements before the **db_check** statement in your test script. You must run the test in Update mode once to capture the expected results set before you run your test in Verify mode.

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.
- 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 340.

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” on page 333. 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 “Messages in the Database Checkpoint Dialog Boxes” on page 336.

---

1 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);
```

2 Create an array 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] = “Denver”;
dbvf1_params[2] = “Monday”;
```

The array replaces the question marks (**?**) in the SQL statement on page 363 with the new values. Follow the guidelines below for adding an array in TSL to parameterize your SQL statements.

3 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 array in TSL.

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.

Specifying a Database

While you are creating a database 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 (Standard database checkpoints only)

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 Microsoft Office 2000)

To create 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 340.

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 366.

---

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 4 on page 331.
   - ➤ To create a custom check on a database, follow the instructions starting at step 6 on page 334.

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

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.

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.

---

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.
5. The Database Checkpoint wizard opens to the **Select conversion file** screen. Follow the instructions in “Selecting a Data Junction Conversion File” on page 342.
6. Continue creating a database checkpoint in WinRunner:
   - ➤ To create a default check on a database, follow the instructions starting at step 4 on page 331.
   - ➤ To create a custom check on a database, follow the instructions starting at step 6 on page 334.

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

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 27, "Generating Functions." For more information about these functions, refer to the *TSL Reference*.

Checking Data in a Database

You use the **db_check** function to create a standard database checkpoint with ODBC (Microsoft Query) and Data Junction. For information on this function, see "Creating a Default Check on a Database" on page 330 and "Creating a Custom Check on a Database" on page 333. For information on parameterizing **db_check** statements, see "Parameterizing Standard Database Checkpoints" on page 363.

Checking Runtime Data in Your Application Against the Data in a Database

You use the **db_record_check** function to create a runtime database record checkpoint with ODBC (Microsoft Query) and Data Junction. For information on this function, see "Creating a Runtime Database Record Checkpoint," on page 316.

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:

1 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.

2 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.

3 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 column 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.

4 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.

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.

18

Checking Bitmaps

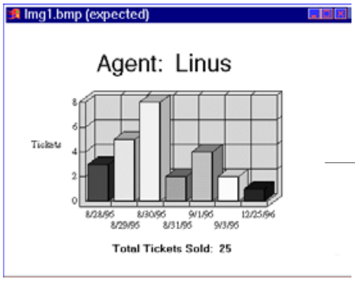
WinRunner enables you to compare two versions of an application being tested by matching captured bitmaps. This is particularly useful for checking non-GUI areas of your application, such as drawings or graphs.

This chapter describes:

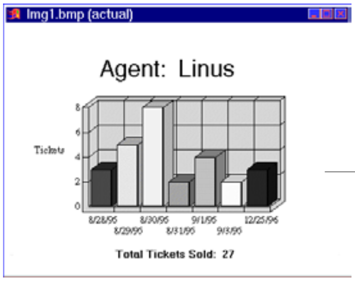
About Checking Bitmaps

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.

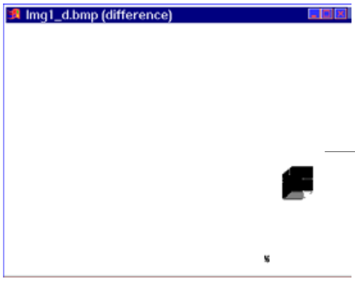
Suppose, for example, your application includes a graph that displays database statistics. You could capture a bitmap of the graph in order to compare it with a bitmap of the graph from a different version of your application. If there is a difference between the graph captured for expected results and 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.

Creating Bitmap Checkpoints

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 **Insert** > **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 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 7, “Editing the GUI Map.”

Your can include your bitmap checkpoint in a loop. If you run your bitmap checkpoint in a loop, the results for each iteration of the checkpoint are displayed in the test results as separate entries. The results of the checkpoint can be viewed in the Test Results window. For more information, see Chapter 34, “Analyzing Test Results.”

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 map files created in XRunner in WinRunner test scripts, see Chapter 9, “Configuring the GUI Map.” For information on using XRunner test scripts recorded in Analog mode, see Chapter 11, “Designing Tests.” For information on using GUI checkpoints created in XRunner, see Chapter 12, “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 "Using Data-Driven Checkpoints and Bitmap Synchronization Points," on page 454.

Handling Differences in Display Drivers

A bitmap checkpoint on identical bitmaps could fail if different display drivers are used when you create the checkpoint and when you run the test, because different display drivers may draw the same bitmap using slightly different color definitions. For example, white can be displayed as RGB (255,255,255) with one display driver and as RGB (231,231,231) with another.

You can configure WinRunner to treat such colors as equal by setting the maximum percentage color difference that WinRunner ignores.

To set the ignorable color difference level:

1. Open *wrun.ini* from the <*WinRunner installation folder*>*dat* folder.
2. Adding the XR_COLOR_DIFF_PRCNT= parameter to the [WrCfg] section.
3. Enter the value indicating the maximum percentage difference to ignore.

In the example described above the difference between each RGB component (255:231) is about 9.4%, so setting the XR_COLOR_DIFF_PRCNT parameter to 10 forces WinRunner to treat the bitmaps as equal:

```
[WrCfg]
XR_COLOR_DIFF_PRCNT=10
```

Setting Bitmap Checkpoint and Capture Options

You can instruct WinRunner to send an e-mail to selected recipients each time a bitmap checkpoint fails and you can instruct WinRunner to capture a bitmap of your window or screen when any checkpoint fails. You set these options in the General Options dialog box.

You can also insert a statement in your script that instructs WinRunner to capture a bitmap of your window or screen based at a specific point in your test run.

To instruct WinRunner to send an e-mail message when a bitmap checkpoint fails:

1. Choose **Tools** > **General Options**. The General Options dialog box opens.
2. Click the **Notifications** category in the options pane. The notification options are displayed.
3. Select **Bitmap checkpoint failure**.
4. Click the **Notifications** > **E-mail** category in the options pane. The e-mail options are displayed.
5. Select the **Active E-mail service** option and set the relevant server and sender information.
6. Click the **Notifications** > **Recipient** category in the options pane. The e-mail recipient options are displayed.
7. Add, remove, or modify recipient details as necessary to set the recipients to whom you want to send an e-mail message when a bitmap checkpoint fails.

   The e-mail contains summary details about the test and the bitmap checkpoint, and gives the file names for the expected, actual, and difference images.

   For more information, see “Setting Notification Options” on page 808.

To instruct WinRunner to capture a bitmap when a checkpoint fails:

1. Choose **Tools** > **General Options**. The General Options dialog box opens.
2. Click the **Run** > **Settings** category in the options pane. The run settings options are displayed.

3 Select **Capture bitmap on verification failure**.

4 Select **Window**, **Desktop**, or **Desktop area** to indicate what you want to capture when checkpoints fail.

5 If you select **Desktop area**, specify the coordinates of the area of the desktop that you want to capture.

When you run your test, the captured bitmaps are saved in your test results folder.

For more information, see "Setting Test Run Options" on page 793.

To capture a bitmap during the test run:

Enter a **win_capture_bitmap** or **desktop_capture_bitmap** statement. Use the following syntax:

```
win_capture_bitmap(image_name [, window, x, y, width, height]);
```

or

```
desktop_capture_bitmap (image_name [, x, y, width, height]);
```

Enter only the desired image name in the statement. Do not include a folder path or extension. The bitmap is automatically stored with a **.bmp** extension in a subfolder of the test results folder.

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

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 **Insert** > **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 **Insert** > **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 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 44, "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 41, "Setting Global Testing Options."

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 **Insert** > **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.

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 44, "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 41, "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 Reference*.

19

Checking Text

WinRunner enables you to read and check text in a GUI object or in any area of your application.

This chapter describes:

- About Checking Text
- Reading Text
- Searching for Text
- Comparing Text
- Teaching Fonts to WinRunner

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**
- read text from a GUI object or window in your application and compare it to expected text, using **obj_check_text** and **win_check_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** property. 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_check_text** function to check that the **Name** edit box in the Flight Reservation window contains the text Kim Smith.

```
set_window ("Flight Reservation", 3);
text_check=obj_check_text ("Name:","Kim Smith");
if (text_check==0)
    report_msg ("The name is correct.");
```

WinRunner can read the visible text from the screen in most applications. 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. For an explanation of when and how to perform this procedure, see "Teaching Fonts to WinRunner" on page 395.

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 can either retrieve the text to a variable, or you can compare the retrieved text with any value you specify.

To retrieve text to a variable, use the **win_get_text**, **obj_get_text**, and **get_text** functions. These functions can be generated automatically, using a **Insert** > **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 **Insert** > **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 **Insert** > **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 and use the image text recognition mechanism. For more information, see "Teaching Fonts to WinRunner" on page 395.

To compare the text in a window or object with an expected text value, use the **win_check_text** or **obj_check_text** functions.

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 **Insert** > **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 **Insert** > **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);
```

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, see Chapter 13, "Working with Web Objects," or refer to the *TSL Reference*.

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 **Insert** > **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 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 395 for more information.

Checking Text in a Window or Object

If you want to compare the value of the text that WinRunner retrieves from an object or window with an expected text value, you can use the **win_check_text**, or **obj_check_text** functions.

Like the **get_text** functions, the **check_text** functions can check all the text in a window or object, or only the text from specified coordinates.

If the expected text and actual text match, the **check_text** functions return the E_OK (0) return code.

When checking the text in a window, use the following syntax:

win_check_text (*window***,** *expected_text* [**,** *x1***,** *y1***,** *x2***,** *y2*] **);**

When checking the text in an object, use the following syntax:

obj_check_text (*object***,** *expected_text* [**,** *x1***,** *y1***,** *x2***,** *y2*] **);**

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

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 27, "Generating Functions." For information about specific functions, refer to the *TSL Reference*.

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.

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 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 Reference.*

Clicking a Specified Text String

The **win_click_on_text** and **obj_click_on_text** functions search for a specified text string in the indicated window or other GUI object, move the screen pointer to the center of the string, and click the string.

The **win_click_on_text** and **obj_click_on_text** functions are Context Sensitive and have similar syntax, as shown:

win_click_on_text (*window, string,* [,x_1,y_1,x_2,y_2] [,*string_def*] [,*mouse_button*] **);**

The *window* or *object* is the window or object to search. The *string* is the text the mouse pointer clicks. The optional *x1,y1,x2,y2* parameters specify the region of the window or object that is searched. The optional *string_def* defines how WinRunner searches for the text. The optional *mouse_button* specifies which mouse button to use.

In the following example, **obj_click_on_text** clicks a topic in an 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 Reference.*

Comparing Text

The **compare_text** function compares two strings, ignoring any differences that you specify. You can use it alone or in conjunction with the **win_get_text** and **obj_get_text** functions.

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

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

The *str1* and *str2* parameters represent the literal strings or string variables to be compared.

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

The **compare_text** function returns 1 when the compared strings are considered the same, and 0 when the strings are considered different. For example, a portion of your test script compares the text string “File” returned by **get_text**. Because the lowercase “l” character has the same shape as the uppercase “I”, you can specify that these two characters be ignored as follows:

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

Teaching Fonts to WinRunner

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 using the Fonts Expert Utility and use the image text recognition mechanism when running your tests.

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. In the **Record** > **Text Recognition** category of the General Options dialog box, select the **Use image-based text recognition** option and enter the font group you created in the **Font group** box.
4. 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.

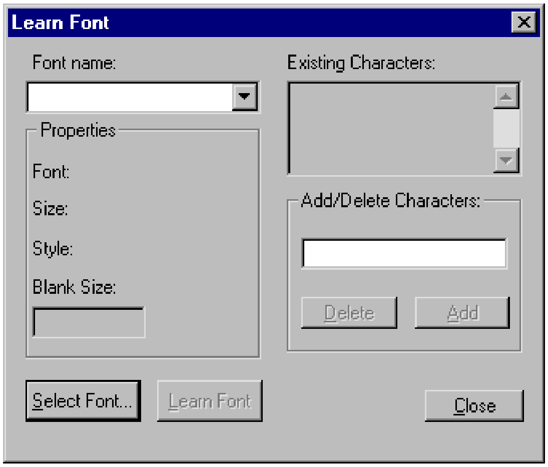
Learning a Font

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 window 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.

Tip: You can ask your programmers for the font name, style, and size.

---

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**.

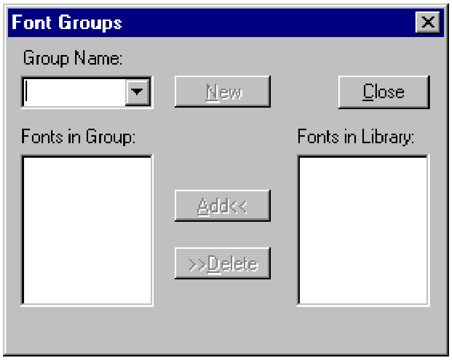
Creating a Font Group

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 window, 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 window, 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 window, 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**.

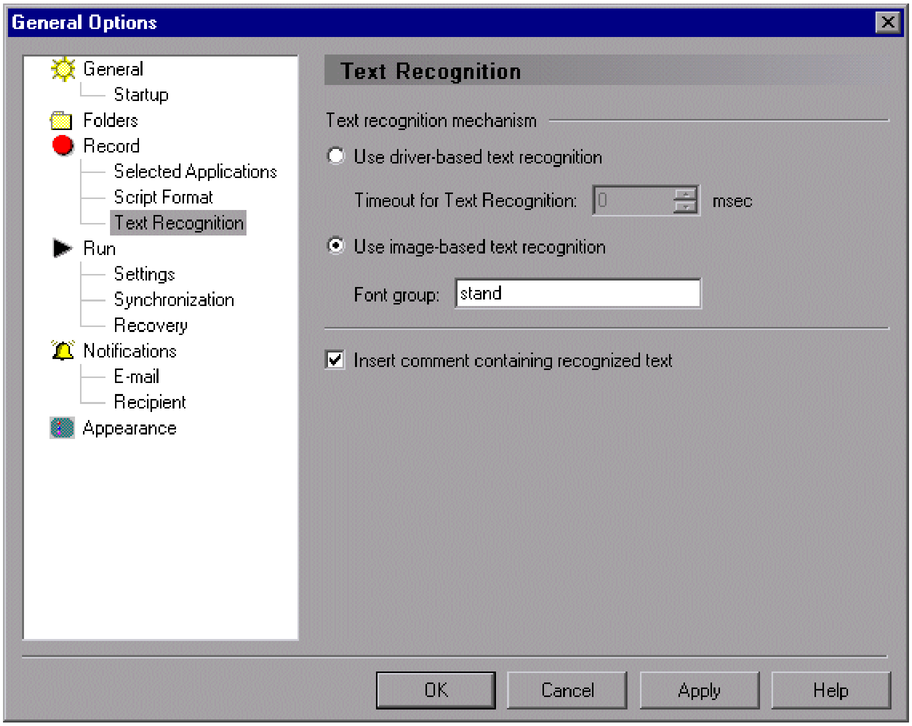
Running Tests on Learned Fonts

In order to instruct WinRunner to use the fonts in your font group, you must use the Image Text Recognition mechanism instead of WinRunner's standard text recognition mechanism and you must activate the font group that includes the fonts your application uses.

To enable WinRunner to recognize learned fonts:

1 Choose **Tools** > **General Options**. The General Options dialog box opens.

2 Choose the **Record** > **Text Recognition** category.



Text Recognition

Text recognition mechanism

○ Use driver-based text recognition

Timeout for Text Recognition: 0 msec

● Use image-based text recognition

Font group: stand

☑ Insert comment containing recognized text

3 Select **Use image-based text recognition.**

4 In the **Font group** box, enter a font group.

5 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 setting text recognition preferences from the General Options dialog box, see Chapter 41, “Setting Global Testing Options.” For more information about using the **setvar** function to choose a font group from within a test script, see Chapter 44, “Setting Testing Options from a Test Script.”

20

Checking Dates

You can use WinRunner to check date operations in your application.

This chapter describes:

- About Checking Dates
- Testing Date Operations
- Testing Two-Character Date Applications
- Setting Date Formats
- Using an Existing Date Format Configuration File
- Checking Dates in GUI Objects
- Checking Dates with TSL
- Overriding Date Settings

About Checking Dates

You can check how your application processes date information. Suppose your application is used by European and North American customers. You may want to check how your application will respond to the different date formats used by these customers.

You can use *aging* to check how your application will react when processing future dates.

Checking date information can also help identify problems if your application was not converted for Year 2000. To check date information in your application, you add checkpoints to your test script. When you add a checkpoint, WinRunner looks for dates in the active window or screen, captures the dates, and stores them as expected results. You can also use aging to simulate how your application will process date information on future dates. When you run a test, a GUI checkpoint compares the expected date to the actual date displayed in the application.

By default, WinRunner's date testing functionality is disabled. Before you can start working with the features described in this chapter you must select the **Enable date operations** check box in the **General** category of the General Options dialog box, save your configuration changes, and restart WinRunner. For additional information, see Chapter 41, "Setting Global Testing Options."

Testing Date Operations

When you check dates in your application, the recommended workflow is as follows:

1. Define the date format(s) currently used in your application, for example, DD/MM/YY, as described in "Setting Date Formats" on page 406 and "Using an Existing Date Format Configuration File" on page 408.
2. Create baseline tests by recording tests on your application. While recording, insert checkpoints that will check the dates in the application. For additional information, see "Checking Dates in GUI Objects" on page 409.
3. Run the tests (in Debug mode) to check that they run smoothly. For more information, see Chapter 33, "Understanding Test Runs."

   If a test incorrectly identifies non-date fields as date fields or reads a date field using the wrong date format, you can override the automatic date recognition on selected fields. For more information, see "Overriding Date Settings" on page 412.
4. Run the tests (in Update mode) to create expected results. For more information, see Chapter 33, "Understanding Test Runs."
5. Run the tests (in Verify mode). If you want to check how your application performs with future dates, you can age the dates before running the test. For more information, see Chapter 33, "Understanding Test Runs."

6 Analyze test results to pinpoint where date-related problems exist in the application. For more information, see Chapter 34, "Analyzing Test Results."

If you change date formats in your application, (e.g. windowing, date field expansion, or changing the date format style from European to North American or vice versa) you should repeat the workflow described above after you redefine the date formats used in your application. For information on windowing and date field expansion, see "Testing Two-Character Date Applications" on page 405. For information on date formats, see "Setting Date Formats" on page 406 and "Using an Existing Date Format Configuration File" on page 408.

Testing Two-Character Date Applications

In the past, programmers wrote applications using two-character fields to manipulate and store dates (for example, '75' represented 1975). Using a two-character date conserved memory and improved application performance at a time when memory and processing power were expensive.

Many of these applications are still in use today, and will continue to be in use well into the 21st century. In industries where age calculation is routinely performed, such as banking and insurance, applications using the two-character date format generate serious errors after December 31, 1999 and must be corrected.

For example, suppose in the year 2001 an insurance application attempts to calculate a person's current age by subtracting his birth date from the current date. If the application uses the two-character date format, a negative age will result (Age = 01 - 30 years = -29).

In order to ensure that applications can accurately process date information in the 21st century, programmers must examine millions of code lines to find date-related functions. Each instance of a two-character date format must be corrected using one of the following methods:

- **Windowing**

  Programmers keep the two-character date format, but define thresholds (cut-year points) that will determine when the application recognizes that a date belongs to the 21st century. For example, if 60 is selected as the threshold, the application recognizes all dates from 0 to 59 as 21st century dates. All dates from 60 to 99 are recognized as 20th century dates.

- **Date Field Expansion**

  Programmers expand two-character date formats to four-characters. For example, "98" is expanded to "1998".

Assessment testing helps you locate date-related problems in your application.

Setting Date Formats

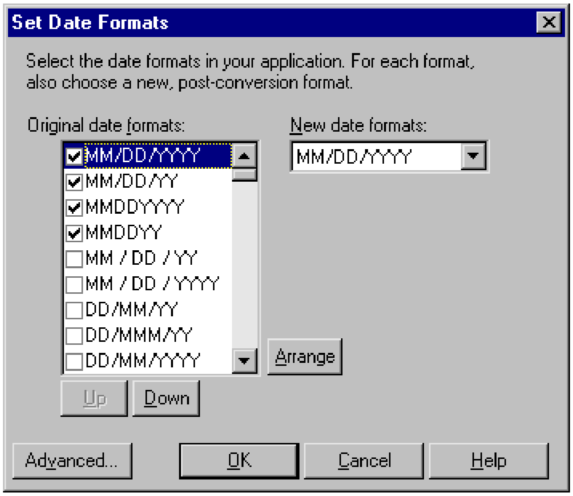
WinRunner supports a wide range of date formats. Before you begin creating tests, you should specify the date formats currently used in your application. This enables WinRunner to recognize date information when you insert checkpoints into a test script and run tests.

By default, WinRunner recognizes the following date formats: MM/DD/YYYY, MM/DD/YY, MMDDYYYY, MMDDYY. In the Set Date Formats dialog box, you can:

- choose which original date formats WinRunner recognizes
- map original date formats to new date formats

To specify date formats:

1 Choose **Tools** > **Date** > **Set Date Formats**. The Set Date Formats dialog box opens.



Set Date Formats

Select the date formats in your application. For each format, also choose a new, post-conversion format.

Original date formats:

- ☑ MM/DD/YYYY
- ☑ MM/DD/YY
- ☑ MMDDYYYY
- ☑ MMDDYY
- □ MM / DD / YY
- □ MM / DD / YYYY
- □ DD/MM/YY
- □ DD/MMM/YY
- □ DD/MM/YYYY

New date formats: MM/DD/YYYY

2 In the **Original date formats** list, select the check box next to each date format used in your application.

3 Click **Arrange** to move all selected date formats to the top of the list. You can also use the **Up** and **Down** buttons to rearrange the formats.

Note that you should move the most frequently-used date format in your application to the top of the list. WinRunner considers the top date format first.

Note that you can also choose from existing date format configuration files to set the date format mapping. For additional information, see “Using an Existing Date Format Configuration File” on page 408.

Using an Existing Date Format Configuration File

WinRunner includes a set of date format configuration files, set for field expansion or windowing preferences, and for European or American styles. You can substitute one of these date format configuration files for the default file used by WinRunner.

To use an existing date format configuration file:

1. In the *<WinRunner installation>\dat* folder, create a backup copy of the existing *y2k.dat* file.
2. Rename one of the files below (in the same location) to y2k.dat, based on your date format preferences:

Configuration File Name	Date Formats
y2k_expn.eur	• **Field expansion:** the converted date field is expanded to four digits. • **European style:** the day followed by the month followed by the year (/DD/MM /YY).
y2k_expn.us	• **Field expansion:** the converted date field is expanded to four digits. • **North American style:** the month followed by the day followed by the year (MM/DD/YY).
y2k_wind.eur	• **Windowing:** the converted date field remains two digits in length. • **European style:** the day followed by the month followed by the year (/DD/MM /YY).
y2k_wind.us	• **Windowing:** the converted date field remains two digits in length. • **North American style:** the month followed by the day followed by the year (MM/DD/YY).

Note that renaming one of these files to *y2k.dat* overwrites your changes to the original *y2k.dat* file.

Checking Dates in GUI Objects

You can use GUI checkpoints to check dates in GUI objects (such as edit boxes or static text fields). In addition you can check dates in the contents of PowerBuilder, Visual Basic, and ActiveX control tables.

When you create a GUI checkpoint, you can use the default check for an object or you can specify which properties to check. When WinRunner's date operations functionality is enabled:

- The default check for edit boxes and static text fields is the date.
- The default check for tables performs a case-sensitive check on the entire contents of a table, and checks all the dates in the table.

Note that you can also use the **Insert** > **GUI Checkpoint** > **For Multiple Objects** command to check multiple objects in a window. For more information about this command, see Chapter 12, "Checking GUI Objects."

Checking Dates with the Default Check

You can use the default check to check dates in edit boxes, static text fields, and table contents.

To check the date in a GUI object:

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

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

2 Click the object containing the date.

3 WinRunner captures the current date and stores it in the test's expected results folder. If you click in a table, WinRunner also captures the table contents. The WinRunner window is restored and a GUI checkpoint is inserted into the test script as an **obj_check_gui** statement. For more information on **obj_check_gui**, refer to the *TSL Reference*.

For additional information on creating GUI checkpoints, see Chapter 12, "Checking GUI Objects,"and Chapter 16, "Checking Table Contents."

Checking Dates Using the Check GUI Dialog Box

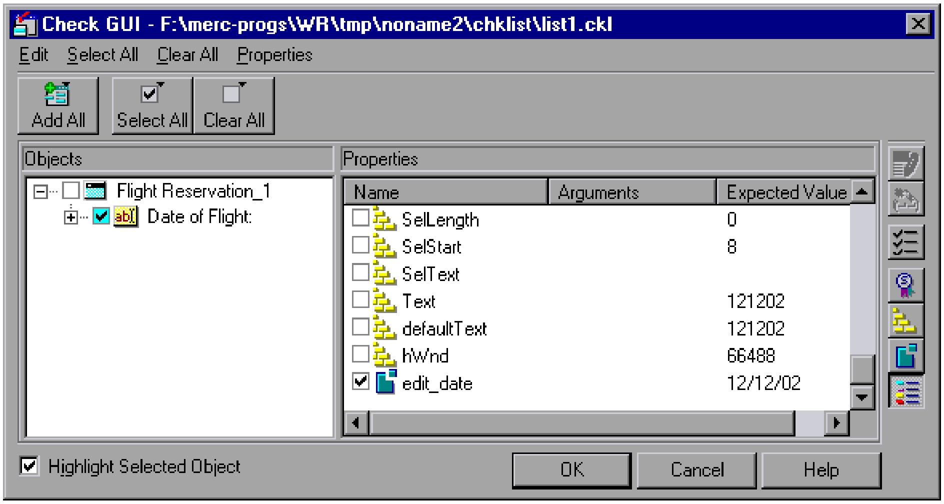
You can create a GUI checkpoint to check a date by specifying which properties of an object to check.

To check dates using the Check GUI dialog box:

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

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

2 Double-click the object containing the date. The Check GUI dialog box opens.



Check GUI - F:\merc-progs\WR\tmp\noname2\chklist\list1.ckl

☐ Flight Reservation_1
☑ Date of Flight:

	Name	Arguments	Expected Value
☐	SelLength		0
☐	SelStart		8
☐	SelText		
☐	Text		121202
☐	defaultText		121202
☐	hWnd		66488
☑	edit_date		12/12/02

☑ Highlight Selected Object

3 Highlight the 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. For more information on selecting properties, see Chapter 12, “Checking GUI Objects,”and Chapter 16, “Checking Table Contents.”

Note that you can edit the expected value of a property. To do so, first select it in the **Properties** column. Next either click the **Edit Expected Value** button, or double-click the value in the **Expected Value** column. For an edit box or a static text field, an edit field opens in the Expected Value column where you can change the value. For a table, the Edit Check dialog box opens. In the **Edit Expected Data** tab, edit the table contents.

5 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 Reference*.

Checking Dates with TSL

You can enhance your recorded test scripts by adding the following TSL **date_** functions:

➤ The **date_calc_days_in_field** function calculates the number of days between two date fields. It has the following syntax:

date_calc_days_in_field (*field_name*$_1$**,** *field_name*$_2$ **);**

➤ The **date_calc_days_in_string** function calculates the number of days between two numeric strings. It has the following syntax:

date_calc_days_in_string (*string*$_1$**,** *string*$_2$ **);**

➤ The **date_field_to_Julian** function translates the contents of a date field to a Julian number. It has the following syntax:

date_field_to_Julian (*date_field* **);**

➤ The **date_is_field** function determines whether a field contains a valid date. It has the following syntax:

date_is_field (*field_name***,** *min_year***,** *max_year* **)**;

➤ The **date_is_string** function determines whether a numeric string contains a valid date. It has the following syntax:

date_is_string (*string***,** *min_year***,** *max_year* **)**;

- The **date_is_leap_year** function determines whether a year is a leap year. It has the following syntax:

  date_is_leap_year (*year* **);**

- The **date_month_language** function sets the language used for month names. It has the following syntax:

  date_month_language (*language* **);**

- The **date_string_to_Julian** function translates the contents of a date string to a Julian number. It has the following syntax:

  date_string_to_Julian (*string* **);**

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

Overriding Date Settings

As you debug your tests, you may want to override how WinRunner identifies or ages specific date fields in your application. You can override the following:

- *Aging of a specific date format*. You can define that a specific date format (for example, MM/DD/YY) will be aged differently than the default aging applied to other date formats.
- *Aging or date format of a specific object*. You can define that a specific object that resembles a date (for example, a catalog number such as 123172) will not be treated as a date object. You can specify that a specific date object (such as a birth date) will not be aged. Or, you can define that a specific object will be assigned a different date format than that of the default.

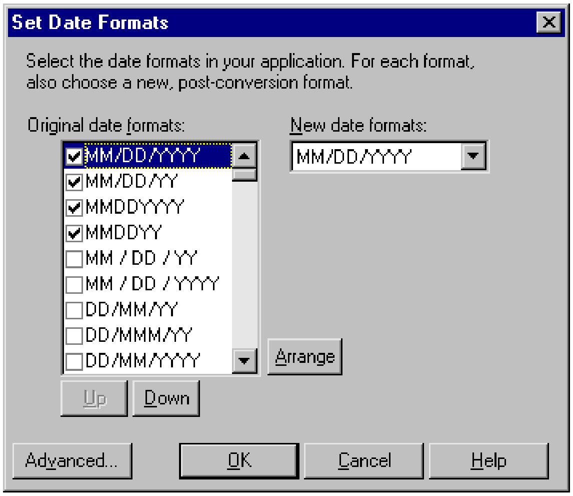
Note: When WinRunner runs tests, it first examines the general settings defined in the Date Operations Run Mode dialog box. Then, it examines the aging overrides for specific date formats. Finally, it considers overrides defined for particular objects.

Overriding Aging of Specific Date Formats

You can override the aging of a specific date format so that it will be aged differently than the default aging setting.

To override the aging of a date format:

1 Choose **Tools** > **Date** > **Set Date Formats**. The Set Date Formats dialog box opens.



Set Date Formats

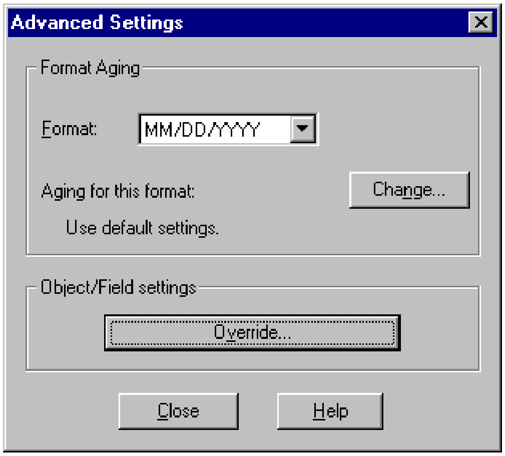
Select the date formats in your application. For each format, also choose a new, post-conversion format.

Original date formats:

☑ MM/DD/YYYY
☑ MM/DD/YY
☑ MMDDYYYY
☑ MMDDYY
□ MM / DD / YY
□ MM / DD / YYYY
□ DD/MM/YY
□ DD/MMM/YY
□ DD/MM/YYYY

New date formats: MM/DD/YYYY

2 Click the **Advanced** button. The Advanced Settings dialog box opens.



Advanced Settings

Format Aging

Format: MM/DD/YYYY

Aging for this format:

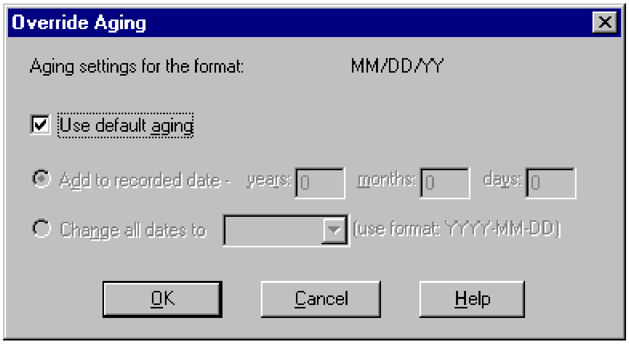
Use default settings.

Object/Field settings

3 In the **Format** list, select a date format.

Note that the Format list displays only the date formats that are checked in the Set Date Formats dialog box.

4 Click **Change**. The Override Aging dialog box opens.



Override Aging

Aging settings for the format: MM/DD/YY

☑ Use default aging

● Add to recorded date - years: 0 months: 0 days: 0

○ Change all dates to [use format: YYYY-MM-DD]

5 Clear the **Use default aging** check box and select one of the following:

- To increment the date format by a specific number of years, months, and days, select the **Add to recorded date** option. To specify no aging for the date format, use the default value of 0.
- To choose a specific date for the selected date format, select **Change all dates to**, and choose a date from the list.

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

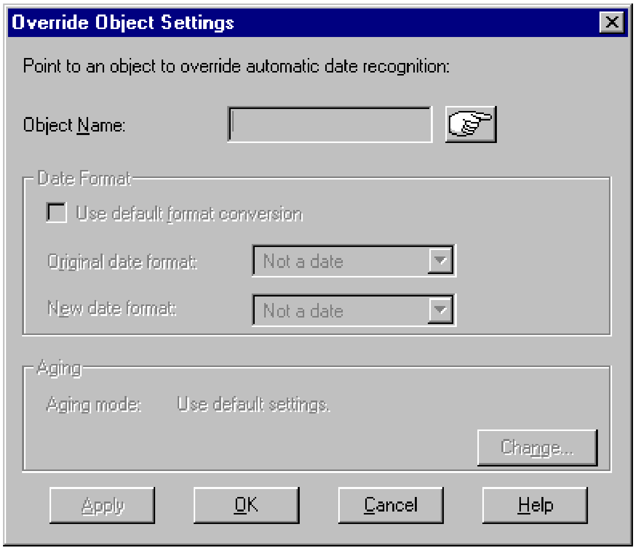
Overriding Aging or Date Format of an Object

For any specific object, you can override the default settings and specify that:

- the object should not be treated like a date object
- the object should be aged differently
- the object should be converted to a different date format

To override settings for an object:

1 Choose **Tools** > **Date** > **Override Object Settings**. The Override Object Settings dialog box opens.



Override Object Settings

Point to an object to override automatic date recognition:

Object Name:

Date Format

□ Use default format conversion

Original date format: Not a date

New date format: Not a date

Aging

Aging mode: Use default settings.

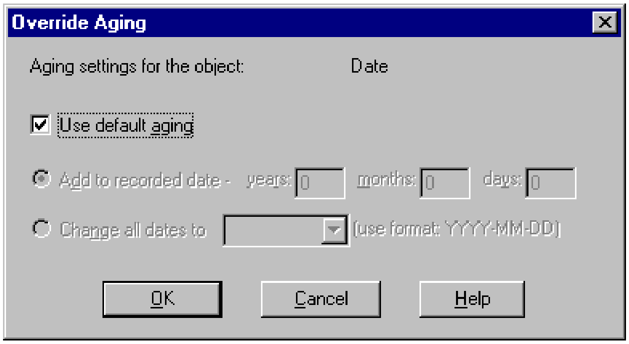
2 Click the pointing hand button and then click the date object.

WinRunner displays the name of the selected date object in the **Object Name** box.

3 To override date format settings or to specify that the object is not a date object, clear the **Use default format conversion** check box and do one of the following:

- To specify that the object should not be treated like a date object, select **Not a date** in the **Original date format** field and in the **New date format** field.
- To override the date format assigned to the object, select the object's original date format and its new date format in the respective fields.

4 To override the aging applied to the object, click **Change**. The Override Aging dialog box opens.



Override Aging

Aging settings for the object: Date

☑ Use default aging

● Add to recorded date - years: 0 months: 0 days: 0

○ Change all dates to (use format: YYYY-MM-DD)

5 Clear the **Use default aging** check box and do one of the following:

- To increment the date format by a specific number of years, months, and days, select the **Add to recorded date** option. To specify no aging for the date format, use the default value of 0.
- To choose a specific date for the selected date format, select **Change all dates to**, and choose a date from the list.

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

7 In the Override Object Settings dialog box, click **Apply** to override additional date objects, or click **OK** to close the dialog box.

Overriding Date Formats and Aging with TSL

You can override dates in a test script using the following TSL functions:

- The **date_age_string** function ages a date string. It has the following syntax:

  date_age_string (*date*, *years*, *month*, *days*, *output* **);**

- The **date_align_day** function ages dates to a specified day of the week or type of day. It has the following syntax:

  date_align_day (*align_mode*, *day_in_week* **);**

- The **date_change_original_new_formats** function overrides the date format for a date object. It has the following syntax:

  date_change_original_new_formats (*object_name*, *original_format*, *new format* [**,** TRUE/FALSE] **);**

- The **date_change_field_aging** function overrides the aging applied to the specified date object. It has the following syntax:

  date_change_field_aging (*field_name*, *aging_type*, *days*, *month*s, *years* **);**

- The **date_set_aging** function ages the test script. It has the following syntax:

  date_set_aging (*format*, *type*, *days*, *months*, *years* **);**

- The **date_set_system_date** function sets the system date and time. It has the following syntax:

  date_set_system_date (*year*, *month*, *day* [**,** *day*, *minute*, *second*] **);**

- The **date_type_mode** function disables overriding of automatic date recognition for all date objects in a GUI application. It has the following syntax:

  date_type_mode (*mode* **);**

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

21

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:

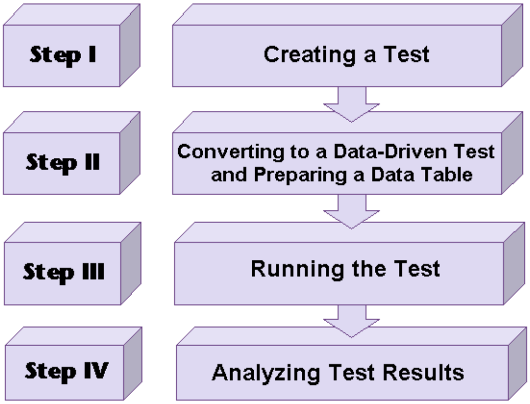
About Creating Data-Driven Tests

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. In each of the ten *iterations*, the test is driven by a different set of data. In order for WinRunner to use data to drive the test, you must substitute fixed values in the test with parameters. The parameters in the test are linked with data 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.

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 two-part step between creating the test and running it: converting the test to a data-driven test and creating a corresponding data table.

The following diagram outlines the stages of the data-driven testing process in WinRunner:

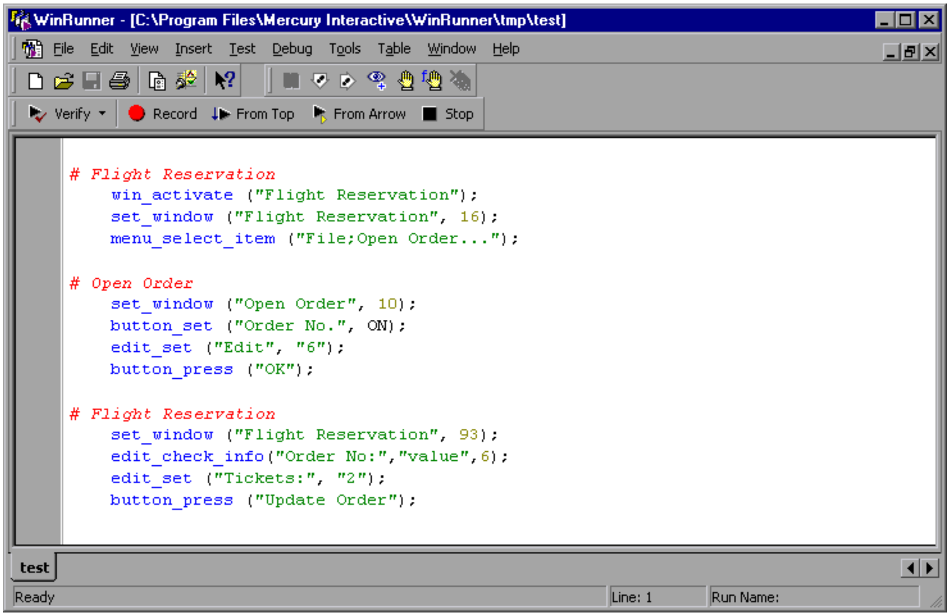


Creating a Basic Test for Conversion

In order to create a data-driven test, you must first create a basic test and then convert 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 and use the **Insert** > **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:



```
# Flight Reservation
    win_activate ("Flight Reservation");
    set_window ("Flight Reservation", 16);
    menu_select_item ("File;Open Order...");

# Open Order
    set_window ("Open Order", 10);
    button_set ("Order No.", ON);
    edit_set ("Edit", "6");
    button_press ("OK");

# Flight Reservation
    set_window ("Flight Reservation", 93);
    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 **Insert** > **GUI Checkpoint** > **For Object/Window** command to insert an **obj_check_gui** statement in your test script. All ***_check_gui** statements contain references to checklists, however, and because 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 **Insert** > **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 12, "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** > **Settings** category 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 41, "Setting Global Testing Options," or Chapter 44, "Setting Testing Options from a Test Script."

You can use the **Insert** > **GUI Checkpoint** > **For Single Property** command to create property checks using the following ***_check_*** functions:

button_check_info

edit_check_info

list_check_info

obj_check_info

scroll_check_info

static_check_info

win_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 27, "Generating Functions."

button_check_state

edit_check_selection

edit_check_text

list_check_item

list_check_selected

scroll_check_pos

static_check_text

For information about specific functions, refer to the *TSL 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 "Using Data-Driven Checkpoints and Bitmap Synchronization Points" on page 454.

Converting a Test to a Data-Driven Test

The procedure for converting a test to a data-driven test is composed of the following main steps:

1 Replacing fixed values in checkpoint statements and in recorded statements with parameters, and creating a data table containing values for the parameters. This is known as *parameterizing* the test.

2 Adding 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 Adding statements to your script that open and close the data table.

4 Assigning 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 Data-Driven Test with the DataDriver Wizard

You can use the DataDriver wizard to convert 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.

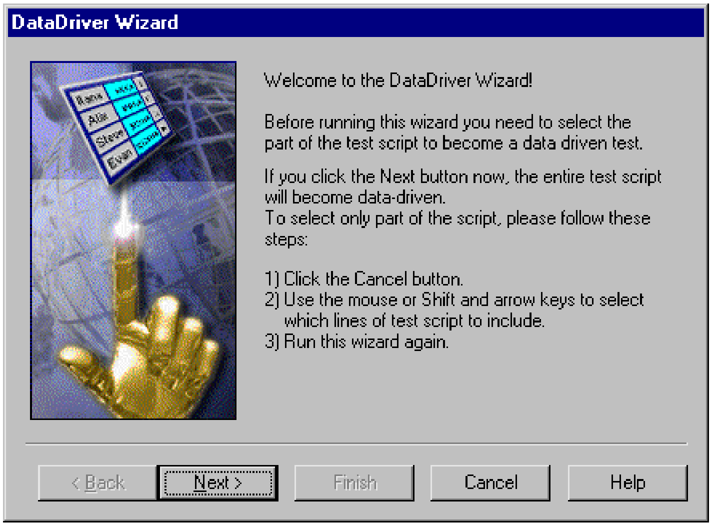
To create a data-driven test:

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.

2 Choose **Table** > **Data Driver Wizard**.

➤ If you selected part of the test script before opening the wizard, proceed to step 3 on page 425.

➤ If you did not select any lines of script, the following screen opens:



DataDriver Wizard

Welcome to the DataDriver Wizard!

Before running this wizard you need to select the part of the test script to become a data driven test.

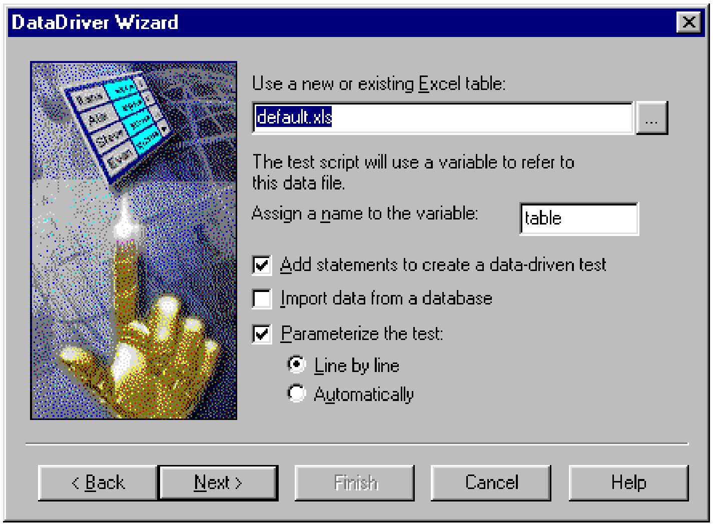
If you click the Next button now, the entire test script will become data-driven.
To select only part of the script, please follow these steps:

1) Click the Cancel button.
2) Use the mouse or Shift and arrow keys to select which lines of test script to include.
3) Run this wizard again.

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:



DataDriver Wizard

Use a new or existing Excel table: default.xls

The test script will use a variable to refer to this data file.

Assign a name to the variable: table

☑ Add statements to create a data-driven test

□ Import data from a database

☑ Parameterize the test:

● Line by line

○ Automatically

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. Accept the default data table for this test, 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.”

At the beginning of a data-driven test, the Excel data table you selected is assigned as the value of the table variable. Throughout the script, only the table variable name is used. This makes it easy for you to assign a different data table to the script at a later time without making changes throughout the script.

Choose from among the following options:

- **Add statements to create a data-driven test:** Automatically adds statements to run your test in a loop: 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 and sample statements, see "Adding Statements to Your Test Script to Open and Close the Data Table and Run Your Test in a Loop" on page 433.

  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 438.

  You can either manually import data from a database by specifying the SQL statement, or you can import data using Microsoft Query or Data Junction. Note that in order to import data using Microsoft Query or Data Junction, 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, the wizard 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.

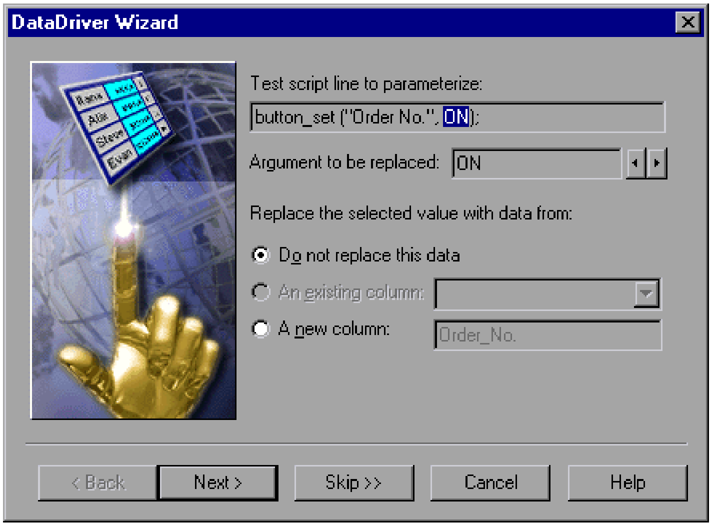
Note: You can also parameterize your test manually. For more information, see "Parameterizing Values in a Test Script" on page 434.

Note: The *ddt_func.ini* file in the *dat* folder of your WinRunner installation 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 29, "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 with "Importing Data from a Database" on page 438. Otherwise, the following wizard screen opens:



Test script line to parameterize: button_set ("Order No.", ON);

Argument to be replaced: ON

Replace the selected value with data from:

● Do not replace this data

○ An existing column:

○ A new column: Order_No.

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 420, 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"));
```

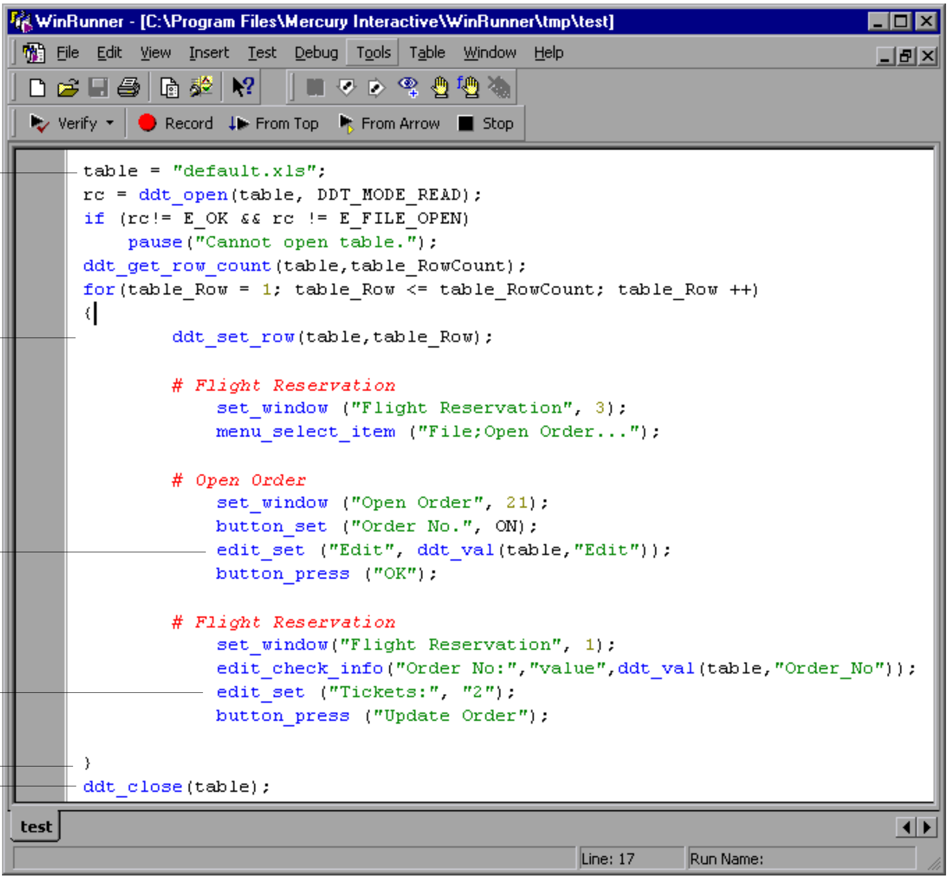
- To parameterize additional lines in your test script, click **Next**. The wizard displays the next line you can parameterize in the test script selection. Repeat the above step for each line in the test script selection that can be parameterized. If there are no more lines in the selection of your 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.

- If you want the data table to open after you close the wizard, select **Show data table now**.
- 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, 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 421 is modified, as shown below:



```
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);

        # Flight Reservation
            set_window ("Flight Reservation", 3);
            menu_select_item ("File;Open Order...");

        # Open Order
            set_window ("Open Order", 21);
            button_set ("Order No.", ON);
            edit_set ("Edit", ddt_val(table,"Edit"));
            button_press ("OK");

        # Flight Reservation
            set_window("Flight Reservation", 1);
            edit_check_info("Order No:","value",ddt_val(table,"Order_No"));
            edit_set ("Tickets:", "2");
            button_press ("Update Order");

}
ddt_close(table);
```

Statements to open data table and run test in a loop

Parameterized statement

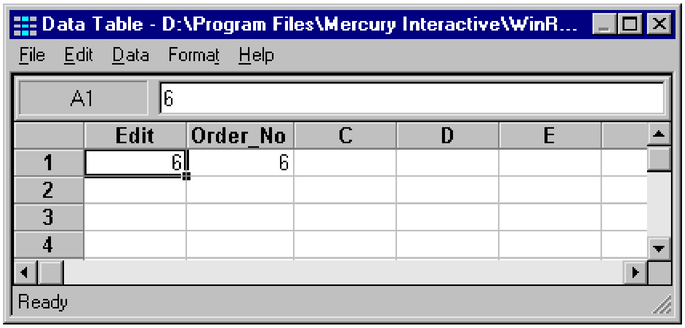
Parameterized property check

End of loop

Statement to close data table

If you open the data table (**Table** > **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.


Data Table - D:\Program Files\Mercury Interactive\WinR...
File Edit Data Format Help
A1
6
Ready

	Edit	Order_No	C	D	E
1	6	6			
2					
3					
4					

Creating a Data-Driven Test Manually

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 466.

```
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.

Note: 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 and later 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.

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 438. For more information on the **ddt_** functions, see "Using TSL Functions with Data-Driven Tests" on page 459 or refer to the *TSL Reference*.

Parameterizing Values in a Test Script

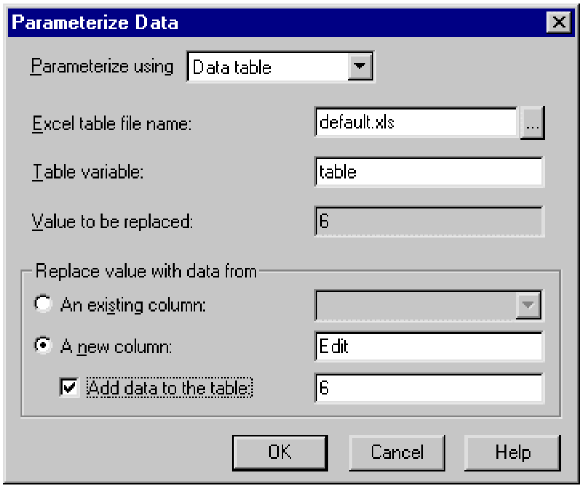
In the sample test script in "Creating a Basic Test for Conversion" on page 420, there are several statements that contain fixed values entered by the user:

```
edit_set("Edit", "6");

edit_check_info("Order No:","value",6);
```

You can use the Parameterize Data dialog box to parameterize the statements and replace the data with parameters.

To parameterize statements using a data table:

1. In your test script, select the first instance in which you have data that you want to parameterize. For example, in the first **edit_set** statement in the test script above, select: "6".
2. Choose **Table** > **Parameterize Data**. The Parameterize Data dialog box opens.
3. In the **Parameterize using** box, select **Data table**.



Parameterize Data

Parameterize using: Data table

Excel table file name: default.xls

Table variable: table

Value to be replaced: 6

Replace value with data from

○ An existing column:

● A new column: Edit

☑ Add data to the table: 6

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.
   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 with quotation marks that was selected in your test script 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.

6 Repeat steps 1 to 5 for each argument you want to parameterize.

For more information on the **ddt_val** function, see "Using TSL Functions with Data-Driven Tests" on page 459 or refer to the *TSL Reference*.

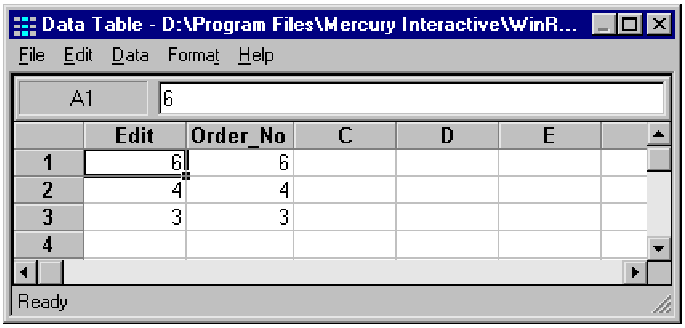
Preparing the Data Table

For each data-driven test, you need to prepare at least one data table. The data table contains the values that WinRunner uses to replace the variables in your data-driven test.

You usually create the data table as part of the test conversion process, either using the DataDriver wizard or the Parameterize Data dialog box. You can also create tables separately in Excel and then link them to the test.

After you create the test, you can add data to the table manually or import it from an existing database.

The following data table displays three sets of data that were entered for the test example described in this chapter. The first set of data was entered using the **Table > Parameterize Data** 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					

- 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, one of which is used for each iteration of a test.

Note: The first character in a column header must be an underscore (_) or a letter. Subsequent characters may be underscores, letters, or numbers.

Adding Data to a Data Table Manually

You can add data to your data table manually by opening the data table and entering values in the appropriate columns.

To add data to a data table manually:

1 Choose **Table > Data Table**. The **Open or Create a Data Table** dialog box opens. Select the data table you specified in the test script to open it, or enter a new name to create a new data table. The data table opens in the data table viewer.

2. Enter data into the table manually.
3. Move the cursor to an empty cell and choose **File** > **Save** from within the data table.

Note: Closing the data table does not automatically save changes to the data table. You must use the **File** > **Save** command from within the data table or a **ddt_save** statement to save the data table. For information on menu commands within the data table, see "Editing the Data Table" on page 438. For information on the **ddt_save** function, see "Using TSL Functions with Data-Driven Tests" on page 459. Note that the data table viewer does not need to be open in order to run a data-driven test.

Importing Data from a Database

In addition to, or instead of, adding data to a data table manually, you can import data from an existing database into your table. You can use either Microsoft Query or Data Junction to import the data. For more information on importing data from a database, see "Importing Data from a Database," on page 444.

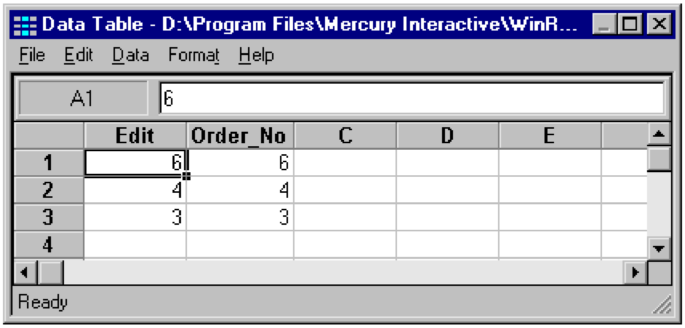
Editing the Data Table

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 **Table** > **Data Table**. The **Open or Create a Data Table** dialog box opens.
3. Select a data table for your test. The data table for the test opens.


Data Table - D:\Program Files\Mercury Interactive\WinR...
File Edit Data Format Help
Ready

A1 6

	Edit	Order_No	C	D	E
1	6	6			
2	4	4			
3	3	3			
4					

4. Use the menu commands described below to edit the data table.
5. Move the cursor to an empty cell and select **File** > **Save** to save your changes.
6. 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*. 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.

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.

File Command	Description
Save	Saves the active data table with its existing name and location. You can save the data table as a Microsoft Excel file or as a tabbed text file.
Save As	Opens the Save As dialog box, which enables you to specify the name and location under which to save the data table. You can save the data table as a Microsoft Excel file or as a tabbed text file.
Import	Imports an existing table file into the data table. This can be a Microsoft Excel 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.
Export	Saves the data table as a Microsoft Excel 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.
Print	Prints the data table.
Print Setup	Enables you to select the printer, the page orientation, and paper size.
Close	Closes the data table. Note that changes are not automatically saved when you close the data table. Use the Save command to save your changes.

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.

Edit Command	Description
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.
Delete	Deletes the current selection. Cells adjacent to the deleted cells are shifted to fill the space left by the vacated cells.
Fill Right	Copies data from the leftmost cell of the selected range of cells to all the cells to the right of it in the range.
Fill Down	Copies data from the top cell of the selected range of cells to all the cells below it in 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.

Format Command	Description
Scientific	Sets format to scientific notation with two decimal places.
Date: (MM/dd/yyyy)	Sets format to Date with the MM/dd/yyyy format.
Time: 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.

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 file.
valid column names	Columns names cannot include spaces. They can include only letters, numbers, and underscores (_).

Importing Data from a Database

In order to import data from an existing database into a data table, you must specify the data to import using the DataDriver wizard. If you selected the **Import data from a database** check box, the DataDriver wizard prompts you to specify the program you will use to connect to the database. You can select either ODBC/Microsoft Query or Data Junction.

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 in the data table with data from an existing column in the database, and 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 overwrites 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

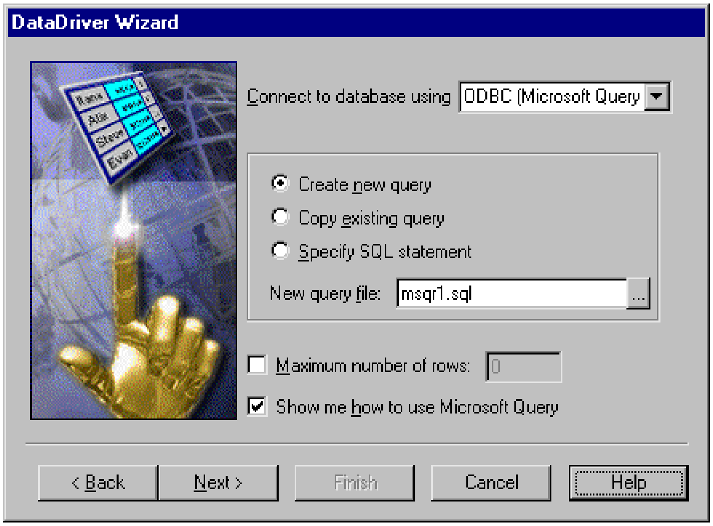
You can use Microsoft Query to choose a data source and define a query within the data source.

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)
- version 2002 (part of Microsoft Office XP)

Setting the Microsoft Query Options

After you select Microsoft Query in the **Connect to database using** option, the following wizard screen opens:



DataDriver Wizard

Connect to database using: ODBC (Microsoft Query

● Create new query
○ Copy existing query
○ Specify SQL statement

New query file: msqr1.sql

□ Maximum number of rows: 0

☑ Show me how to use Microsoft Query

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 446.
- **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 447.
- **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 448.
- **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 Reference.*
- **Show me how to use Microsoft Query:** Displays an instruction screen.

Creating a New Source Query File

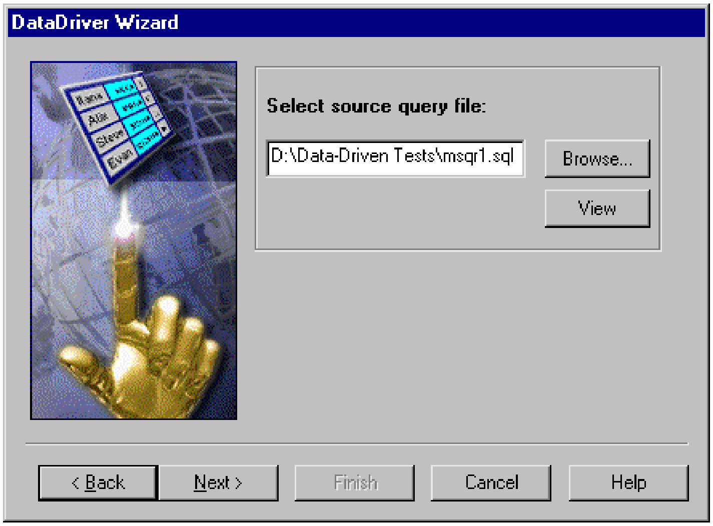
Microsoft Query opens if you chose **Create new query** in the last step. Choose a new or existing data source, define a query, and when you are done:

- 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 converting your test to a data-driven test. For additional information, see step 4 on page 428.

Selecting a Source Query File

The following screen opens if you chose **Copy existing query** in the last step.



Select source query file:

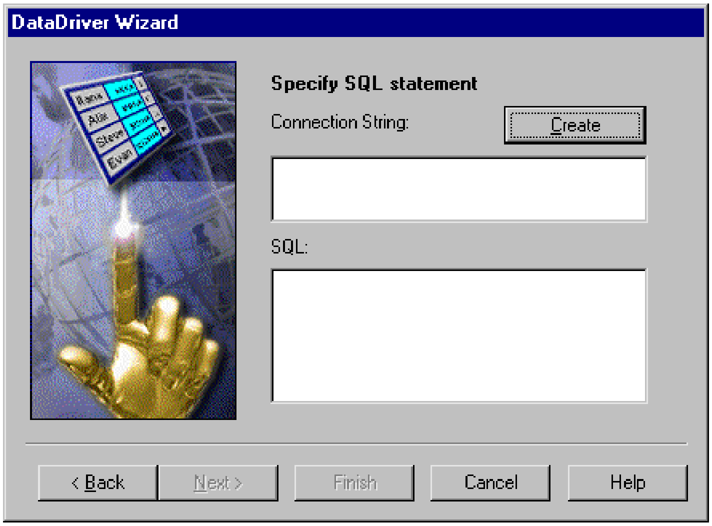
D:\Data-Driven Tests\msqr1.sql

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 428.

Specifying an SQL Statement

The following screen opens if you chose **Specify SQL statement** in the last step.



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 428.

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.

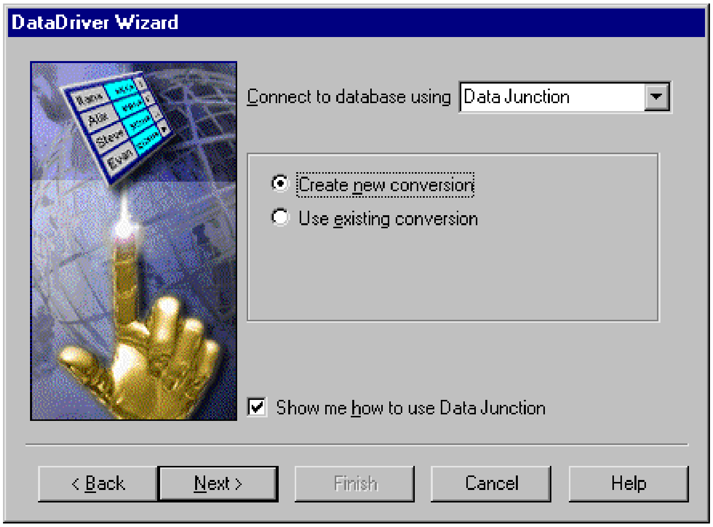
Importing Data from a Database Using Data Junction

You can use Data Junction to create a conversion file that converts a database to a target text file.

Note that WinRunner supports versions 6.5 and 7 of Data Junction.

Setting the Data Junction Options

If Data Junction is installed on your machine, the following wizard screen opens once you choose to import data from a Data Junction database:



DataDriver Wizard

Connect to database using: Data Junction

● Create new conversion
○ Use existing conversion

☑ Show me how to use Data Junction

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 450.

- **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 451.

- **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.

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.

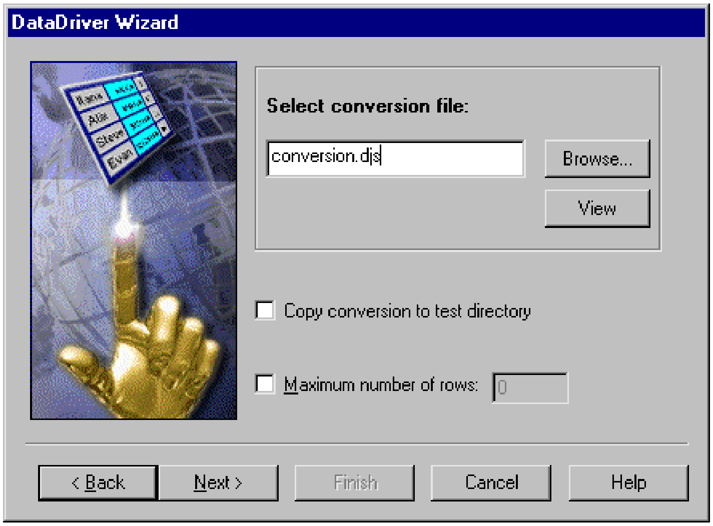
---

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.
5. The DataDriver wizard opens to the **Select conversion file** screen. Follow the instructions in "Selecting a Data Junction Conversion File" on page 451.

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.



DataDriver Wizard

Select conversion file:

conversion.djs

Browse...
View
□ Copy conversion to test directory
□ Maximum number of rows: 0
< Back
Next >
Finish
Cancel
Help

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 Data Junction Conversion Manager if you want to view the file.

You can also choose from the following options:

- **Copy conversion to test directory:** 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 428.

Running and Analyzing Data-Driven Tests

You run and analyze data-driven tests much the same as for any WinRunner test. The following two sections describe these two procedures.

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 33, "Understanding Test Runs."

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 438. For information on the **ddt_update_from_db** function, see "Using TSL Functions with Data-Driven Tests" on page 459 or refer to the *TSL Reference*.

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 a certain event occurs during each iteration, then the test results will record a separate result for the event for each iteration.

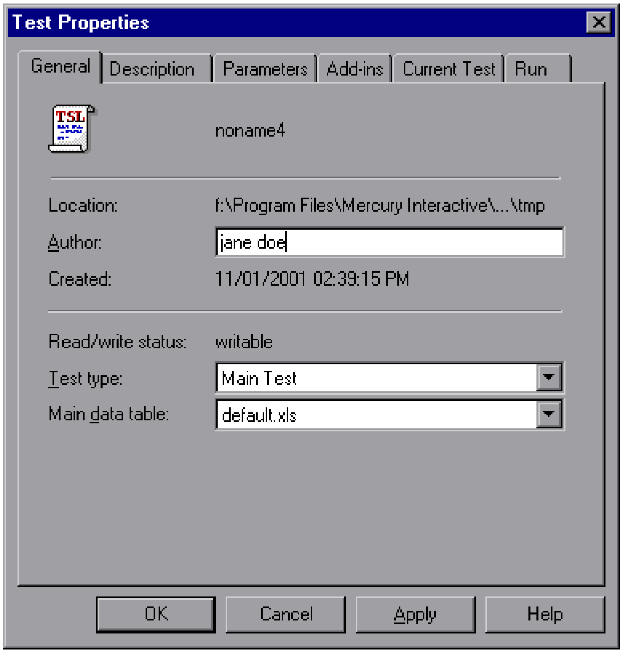
For example, if you inserted a **ddt_report_row** statement in your test script, then WinRunner prints a row of the data table to the test results. Each iteration of a **ddt_report_row** statement in your test script creates a line in the Test Log table in the Test Results window, identified as "Table 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 465 or refer to the *TSL Reference*. For more information on viewing test results, see Chapter 34, "Analyzing Test Results."

Assigning the Main Data Table for a Test

You can easily set the main data table for a test in the **General** tab of the Test Properties dialog box. The main data table is the table that is selected by default when you choose **Tools** > **Data Table** or open the DataDriver wizard.

To assign the main data table for a test:

1 Choose **File** > **Test Properties** and click the **General** tab.



noname4

Location: f:\Program Files\Mercury Interactive\...\tmp

Author: jane doe

Created: 11/01/2001 02:39:15 PM

Read/write status: writable

Test type: Main Test

Main data table: default.xls

2 Choose the data table you want to assign from the **Main data table** list.

All data tables that are stored in the test folder are displayed in the list.

3 Click **OK**. The data table you select is assigned as the new main data table.

Note: If you open a different data table after selecting the main data table from the Test Properties dialog box, the last data table opened becomes the main data table.

Using 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 **Insert** > **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 for Conversion" on page 420. For information on checking a single property of an object, see Chapter 12, "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.

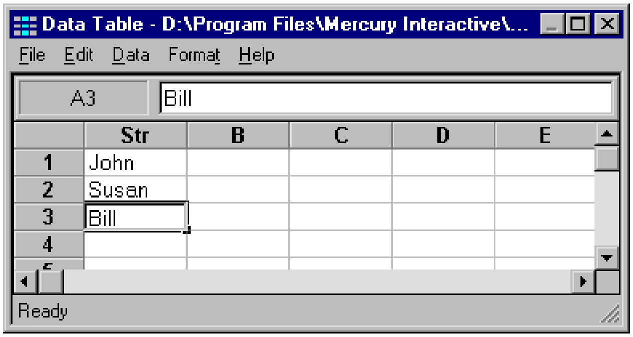
```
set_window ("Untitled - Notepad", 12);
menu_select_item ("Search;Find...");
set_window ("Find", 5);
edit_set ("Find what:", "John");
button_press ("Find Next");
set_window("Notepad", 10);
obj_check_gui("Message", "list1.ckl", "gui1", 1);
win_check_bitmap("Notepad", "img1", 5, 30, 23, 126, 45);
obj_wait_bitmap("Message", "img2", 13);
set_window ("Notepad", 5);
button_press ("OK");
set_window ("Find", 4);
button_press ("Cancel");
```

2 Use the DataDriver wizard (**Table** > **Data Driver 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 “Creating a Data-Driven Test with the DataDriver Wizard,” on page 423. Alternatively, you can make these changes to the test script manually. For additional information, see “Creating a Data-Driven Test Manually,” on page 432.

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:


Data Table - D:\Program Files\Mercury Interactive\...
File Edit Data Format Help
Ready

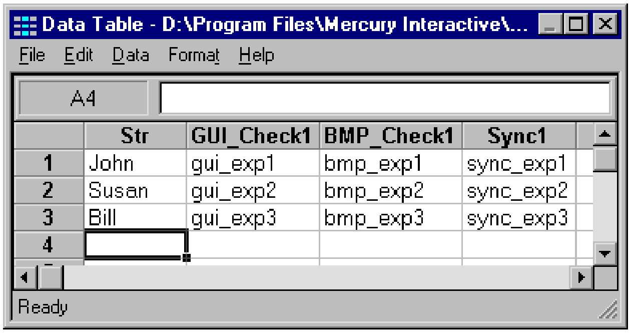
A3 Bill

	Str	B	C	D	E
1	John				
2	Susan				
3	Bill				
4					

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 would 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:


Data Table - D:\Program Files\Mercury Interactive\...
File Edit Data Format Help
Ready

A4

	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				

4 Choose **Table** > **Parameterize Data** 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 "Parameterizing Values in a Test Script," on page 434. 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 27, “Generating Functions.” For more information about these functions, refer to the *TSL Reference*.

Note: You must use the **ddt_open** function to open the data table before you use any other **ddt_** functions. You must use the **ddt_save** function to save the data table, and use the **ddt_close** function to close the data table.

Opening a Data Table

The **ddt_open** function creates or opens the specified data table. The data table is a Microsoft Excel 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_name* [**,** *mode*] **);**

The *data_table_name* is the name of the data table. 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_name* **);**

The *data_table_name* is the name of the data table.

Note that **ddt_save** does not close the data table. Use the **ddt_close** function, described below, to close the data table.

Closing a Data Table

The **ddt_close** function closes the specified data table. This function has the following syntax:

ddt_close (*data_table_name* **);**

The *data_table_name* is the name of the data table.

Note that **ddt_close** does not save changes made to the data table. Use the **ddt_save** function, described above, to save changes before closing the data table.

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.

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_name* [**,** *show_flag*] **);**

The *data_table_name* 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_name*, *out_rows_count* **);**

The *data_table_name* is the name of the data table. 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_name* **);**

The *data_table_name* is the name of the data table.

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_name*, *row* **);**

The *data_table_name* is the name of the data table. 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_name*, *parameter*, *value* **);**

The *data_table_name* is the name of 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.

Notes: You can only use this function if the data table was opened in DDT_MODE_READWRITE (read or write mode).

To save the new contents of the table, add a **ddt_save** statement after the **ddt_set_val** statement. At the end of your test, use a **ddt_close** statement to close the table.

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_name*, *row*, *parameter*, *value* **);**

The *data_table_name* is the name of the data table. 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.

Notes: You can only use this function if the data table was opened in DDT_MODE_READWRITE (read or write mode).

To save the new contents of the table, add a **ddt_save** statement after the **ddt_set_val** statement. At the end of your test, use a **ddt_close** statement to close the table.

Retrieving the Active Row of a Data Table

The **ddt_get_current_row** function retrieves the active row in the specified data table. This function has the following syntax:

ddt_get_current_row (*data_table_name*, *out_row* **);**

The *data_table_name* is the name of the data table. 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_name*, *parameter* **);**

The *data_table_name* is the name of the data table. 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_name*, *params_list*, *params_num* **);**

The *data_table_name* is the name of the data table. 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_name*, *parameter* **);**

The *data_table_name* is the name of the data table. 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_name*, *row_number*, *parameter* **);**

The *data_table_name* is the name of the data table. 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_name* **);**

The *data_table_name* is the name of the data table.

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_name*, *file,out_row_count*
    [**,** *max_rows*] **);**

The *data_table_name* is the name of the data table.

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.

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 438.

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 must save and close the data table in one test before opening it in another test.

- Before you run a data-driven test, you should look through it to see if there are any elements that may cause a conflict in a data-driven test. The DataDriver and Parameterization wizards find all fixed values in selected checkpoints and recorded statements, but they do not check for things such as object labels that also may vary based on external input. There are two ways to solve most of these conflicts:
  - Use a regular expression to enable WinRunner to recognize objects based on a portion of its physical description.
  - Use the GUI Map Configuration dialog box to change the physical properties that WinRunner uses to recognize the problematic object.
- 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 459.
- 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 459.

- You can add **tl_step** or other reporting statements within the parameterized loop of your test so that you can see the result of the data used in each iteration.
- 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 "Using Data-Driven Checkpoints and Bitmap Synchronization Points" on page 454.
- 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.

22

Synchronizing the Test Run

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:

- About Synchronizing the Test Run
- Waiting for Objects and Windows
- Waiting for Property Values of Objects and Windows
- Waiting for Bitmaps of Objects and Windows
- Waiting for Bitmaps of Screen Areas
- Tips for Synchronizing Tests

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 instructs WinRunner to suspend running the test until the application being tested is ready, and then to continue the test.

There are three kinds of synchronization points: object/window synchronization points, property value synchronization points, and bitmap synchronization points.

- When you want WinRunner to wait for an object or a window to appear, you create an object/window synchronization point.

- 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 **Insert** > **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: All **wait** and **wait_info** functions are implemented in milliseconds, so they do not affect how the test runs.

Waiting for Objects and Windows

You can create a synchronization point that instructs WinRunner to wait for a specified object or window to appear. For example, you can tell WinRunner to wait for a window to open before performing an operation within that window, or you may want WinRunner to wait for an object to appear in order to perform an operation on that object.

WinRunner waits no longer than the default timeout setting before executing the subsequent statement in a test script. You can set this default timeout setting in a test script by using the *timeout_msec* testing option with the **setvar** function. For more information, see Chapter 44, “Setting Testing Options from a Test Script.” You can also set this default timeout setting globally using the **Timeout for checkpoints and CS statements** box in the **Run** > **Settings** category of the General Options dialog box. For more information, see Chapter 41, “Setting Global Testing Options.”

You use the **obj_exists** function to create an object synchronization point, and you use the **win_exists** function to create a window synchronization point. These functions have the following syntax:

obj_exists (*object* [**,** *time*] **);**

win_exists (*window* [**,** *time*] **);**

The *object* is the logical name of the object. The object may belong to any class. The *window* is the logical name of the window. The *time* is the amount of time (in seconds) that is added to the default timeout setting, yielding a new maximum wait time before the subsequent statement is executed.

You can use the Function Generator to insert this function into your test script or you can insert it manually. For information on using the Function Generator, see Chapter 27, "Generating Functions." For more information on these functions and examples of usage, refer to the *TSL Reference*.

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 **Insert** > **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 44, "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** > **Settings** category of the General Options dialog box. For information about setting this testing option globally, see Chapter 41, "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 7, "Editing the GUI Map," and Chapter 25, "Using Regular Expressions."

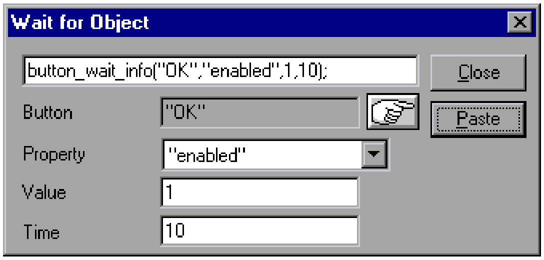
During recording, when you capture an object in a window other than the active window, WinRunner automatically generates a **set_window** statement.

To insert a property value synchronization point:

1 Choose **Insert** > **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.



Wait for Object

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

Button: "OK"

Property: "enabled"

Value: 1

Time: 10

4 Specify the parameters of the property check to be carried out on the window or object, as follows:

- **Window or <Object type>:** 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.

- **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.
- **Time**: 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 44, "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** > **Settings** category of the General Options dialog box. For more information, see Chapter 41, "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.

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 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 27, "Generating Functions." For more information about working with these functions, refer to the *TSL 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 **Insert** > **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 44, "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** > **Settings** category of the General Options dialog box. For information about setting this testing option globally, see Chapter 41, “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 7, “Editing the GUI Map,” and Chapter 25, “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 **Insert** > **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 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 44, "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** > **Synchronization** and **Run** > **Settings** categories of the General Options dialog box. For more information about setting these testing options globally, see Chapter 41, "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 **Insert** > **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 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.

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 44, "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** > **Settings** category of the General Options dialog box. For information about setting this testing option globally, see Chapter 41, "Setting Global Testing Options."

To define a bitmap synchronization point for an area of the screen:

1 Choose **Insert** > **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 Reference*.

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 44, "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** > **Settings** and **Run** > **Synchronization** categories in the General Options dialog box. For more information about setting these testing options globally, see Chapter 41, "Setting Global Testing Options."

Tips for Synchronizing Tests

- **Stopping or pausing a test:** You can stop or pause a test that is waiting for a synchronization statement by using the PAUSE or STOP softkeys. For information on using softkeys, see "Activating Test Creation Commands Using Softkeys" on page 158.
- **Recording in Analog mode:** 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 Reference*.
- **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 "Using Data-Driven Checkpoints and Bitmap Synchronization Points," on page 454.

23

Defining and Using Recovery Scenarios

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

This chapter describes:

About Defining and Using Recovery Scenarios

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

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

There are two types of recovery scenarios:

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

A recovery scenario has two main components:

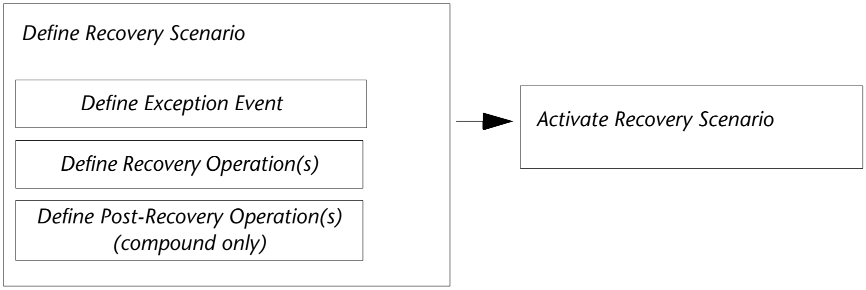
- **Exception Event**: The event that interrupts your test run.
- **Recovery Operation(s)**: The operation(s) that terminate the interruption.

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

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

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

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


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

Recovery scenarios apply only to Windows events. You can also define Web exceptions and handler functions. For more information, see Chapter 24, "Handling Web Exceptions."

Defining Simple Recovery Scenarios

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

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

You can also define simple recovery scenarios using TSL statements. For more information, see "Working with Recovery Scenarios in Your Test Script," on page 520.

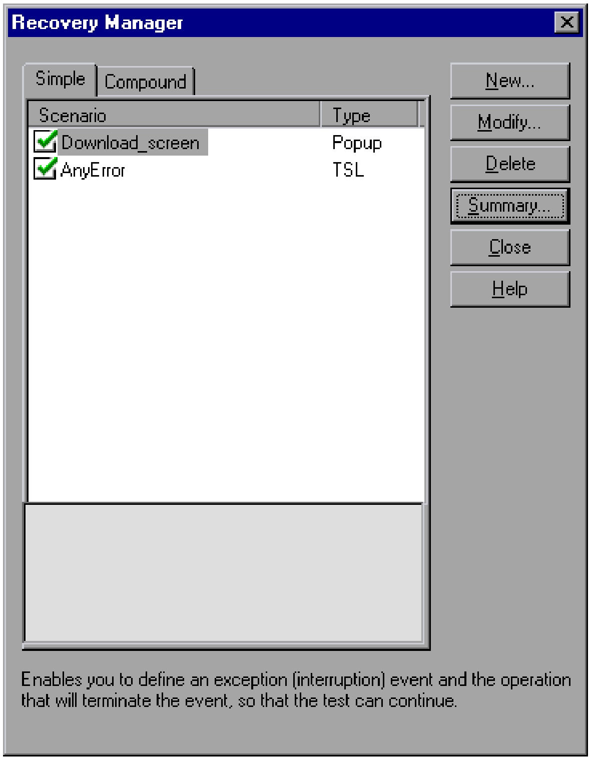
Notes:

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

The first time you use the Recovery Manager to add, modify, or delete a recovery scenario, WinRunner prompts you to select a new recovery scenarios file. For more information, see "Working with Recovery Scenarios Files," on page 516.

To create a simple recovery scenario

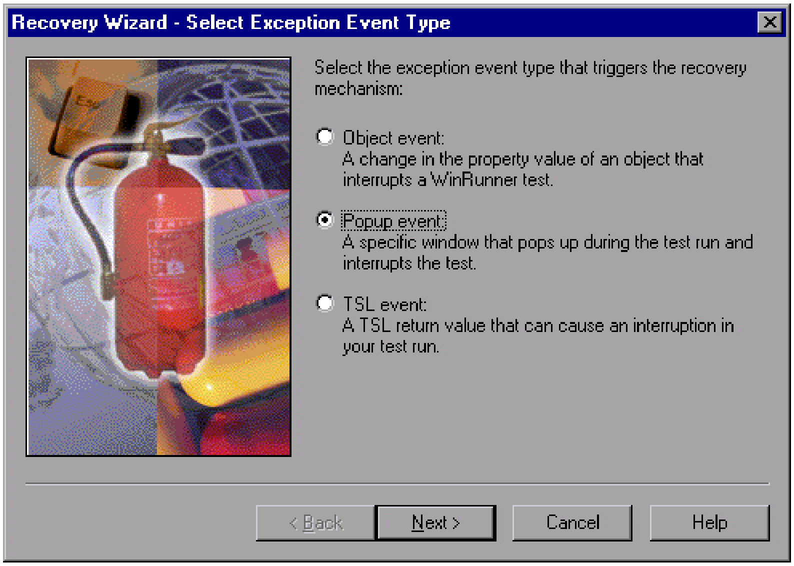
1 Choose **Tools** > **Recovery Manager**. The Recovery Manager opens.



Scenario	Type
☑ Download_screen	Popup
☑ AnyError	TSL

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

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



Recovery Wizard - Select Exception Event Type

Select the exception event type that triggers the recovery mechanism:

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

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

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

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

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

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

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

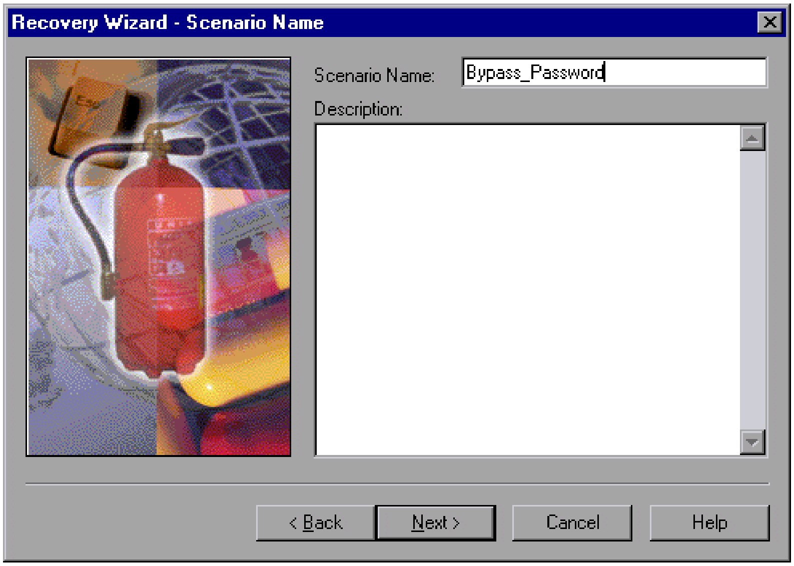
  For example, suppose part of your test includes clicking on a Print button to send a generated graph to the printer, and a message box opens indicating that the printer is out of paper. Your test cannot continue until you close the message box.

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

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

Click **Next**.

4 The Scenario Name screen opens.



Recovery Wizard - Scenario Name

Scenario Name: Bypass_Password

Description:

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

Click **Next**.

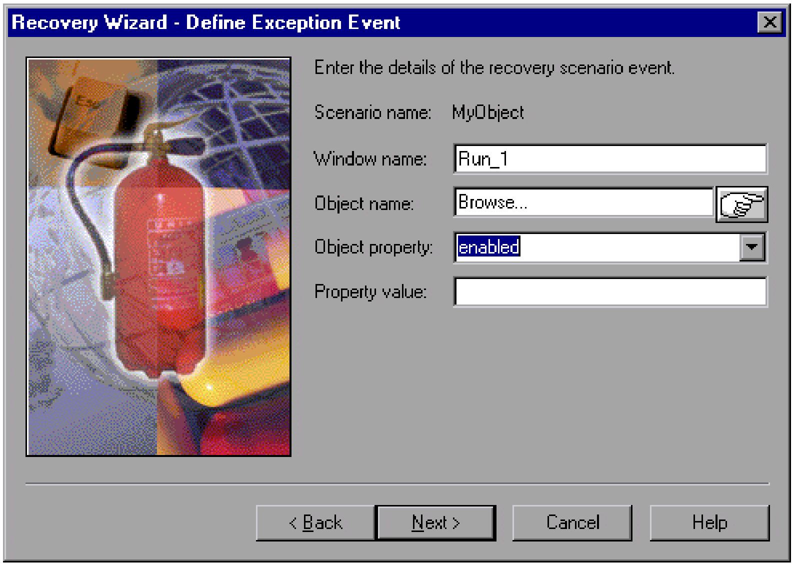
5 The Define Exception Event screen opens. The options in the screen vary based on the type of event you selected in step 3.

For information on defining object events, see page 487.

For information on defining pop-up events, see page 489.

For information on defining TSL events, see page 490.

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



Recovery Wizard - Define Exception Event

Enter the details of the recovery scenario event.

Scenario name: MyObject

Window name: Run_1

Object name: Browse...

Object property: enabled

Property value:

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

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

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

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

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

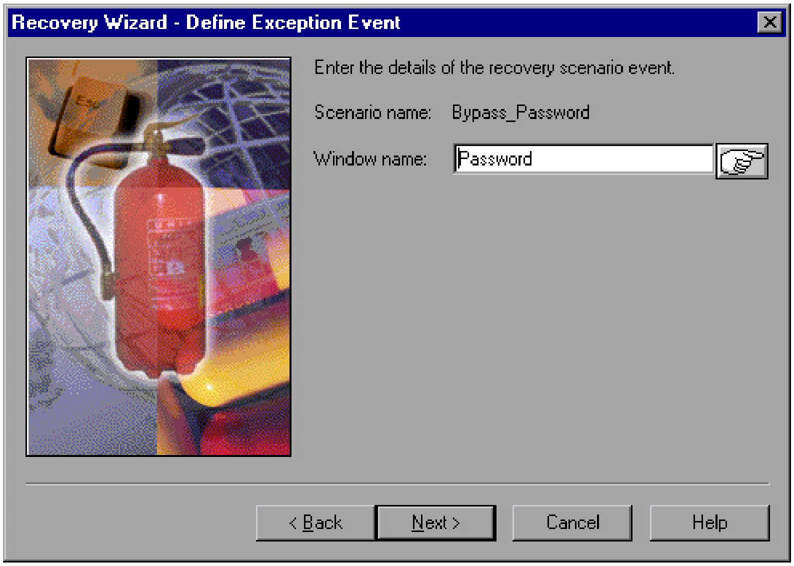
Note: You cannot specify a property that is part of the object's physical description.

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

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

Click **Next** and proceed to step 6 on page 491.

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



Recovery Wizard - Define Exception Event

Enter the details of the recovery scenario event.

Scenario name: Bypass_Password

Window name: Password

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

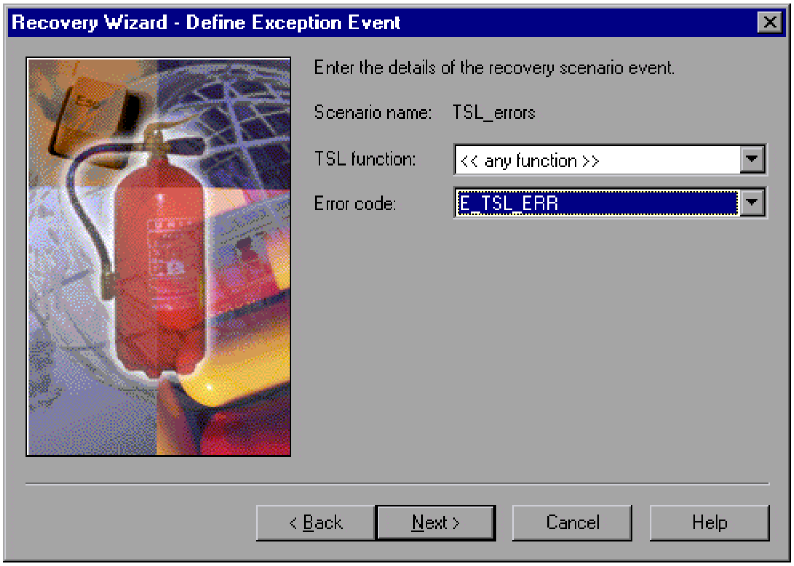
  If the window is not already saved in the GUI Map and you use the pointing hand to identify the window, WinRunner automatically adds it to the active GUI Map. If the window is not already saved in the GUI Map and you type the name manually, WinRunner identifies the pop-up exception when a pop-up window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

Note: If you want to employ the **Click button** recovery operation, then the pop-up window you define must be saved in the GUI Map. If you type the window name manually, you must also add the window to the GUI Map. For more information about recovery operations, see page 491.

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

Click **Next** and proceed to step 6 on page 491.

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



Recovery Wizard - Define Exception Event

Enter the details of the recovery scenario event.

Scenario name: TSL_errors

TSL function: << any function >>

Error code: E_TSL_ERR

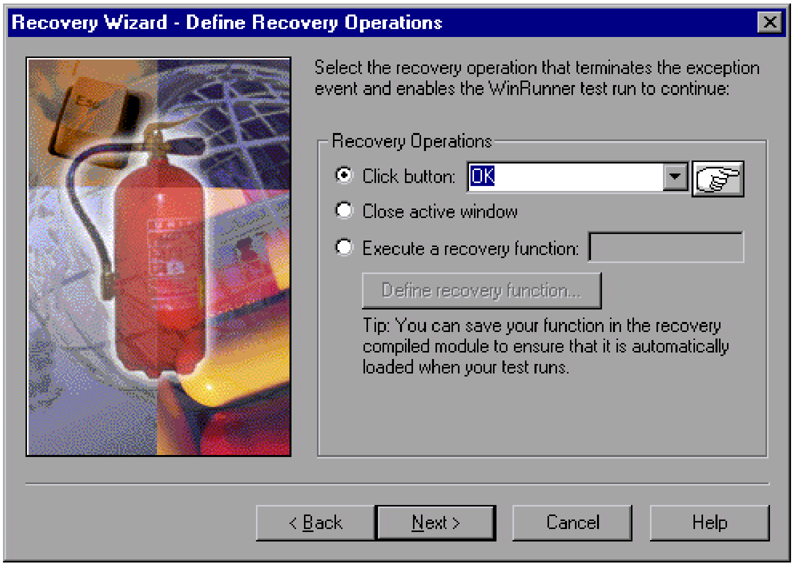
➤ **TSL function:** Select the TSL function for which you want to define the exception event. Select a TSL function from the list. WinRunner detects the exception only when the selected TSL function returns the code selected in the Error code box.

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

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

  Click **Next**.

6 The Define Recovery Operations screen opens.



Recovery Wizard - Define Recovery Operations

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

Recovery Operations

● Click button: OK

○ Close active window

○ Execute a recovery function:

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

Select one of the following recovery options:

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

Notes:

This option is available only for pop-up exceptions.

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

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

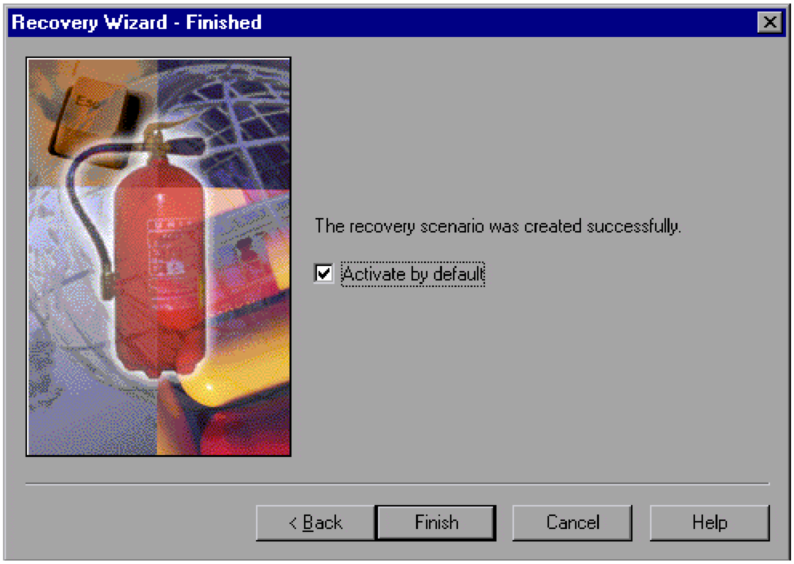
Note: WinRunner uses the (TSL) **win_close** mechanism to close the window. If the **win_close** function cannot close the window, the recovery scenario cannot close the window. In these situations, use the **Click button** or **Execute a recovery function** options instead.

- **Execute a recovery function**: Instructs WinRunner to run the specified function when the exception event occurs. You can specify an existing function or click **Define recovery function** to define a new function. For more information on defining recovery functions, see "Defining Recovery Scenario Functions," on page 509.

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

Click **Next**.

7 The Finished screen opens.



Recovery Wizard - Finished

The recovery scenario was created successfully.

☑ Activate by default

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

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

For information on other ways to activate or deactivate a recovery scenario, see “Activating and Deactivating Recovery Scenarios,” on page 514 and “Working with Recovery Scenarios in Your Test Script,” on page 520.

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

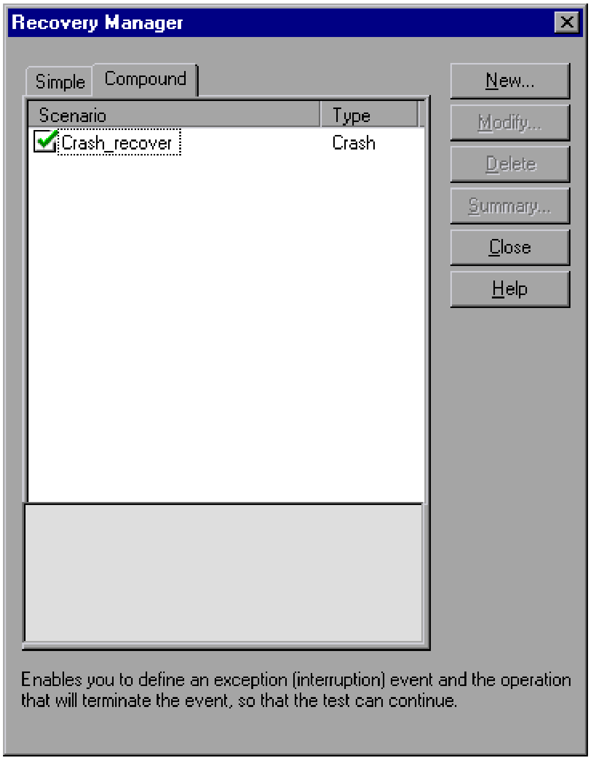
Defining Compound Recovery Scenarios

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

To create a compound recovery scenario

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

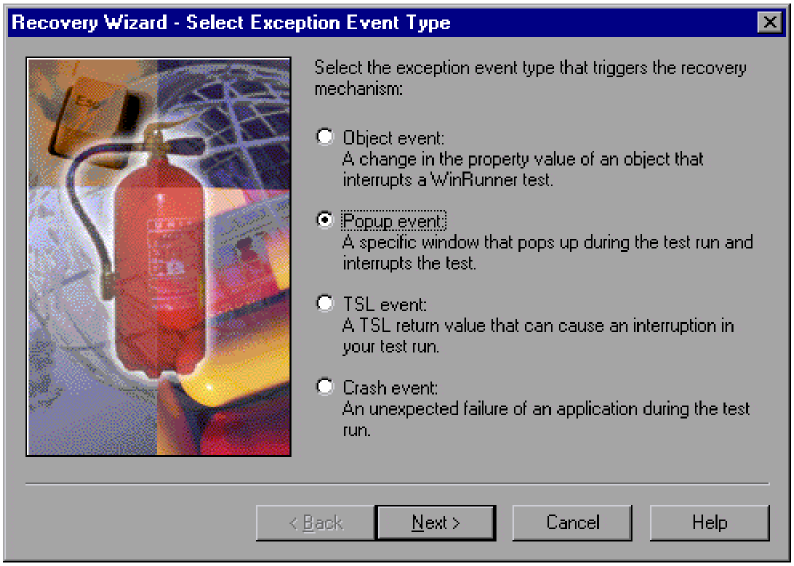
2. Click the **Compound** tab.



Scenario	Type
☑ Crash_recover	Crash

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

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



Recovery Wizard - Select Exception Event Type

Select the exception event type that triggers the recovery mechanism:

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

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

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

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

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

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

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

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

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

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

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

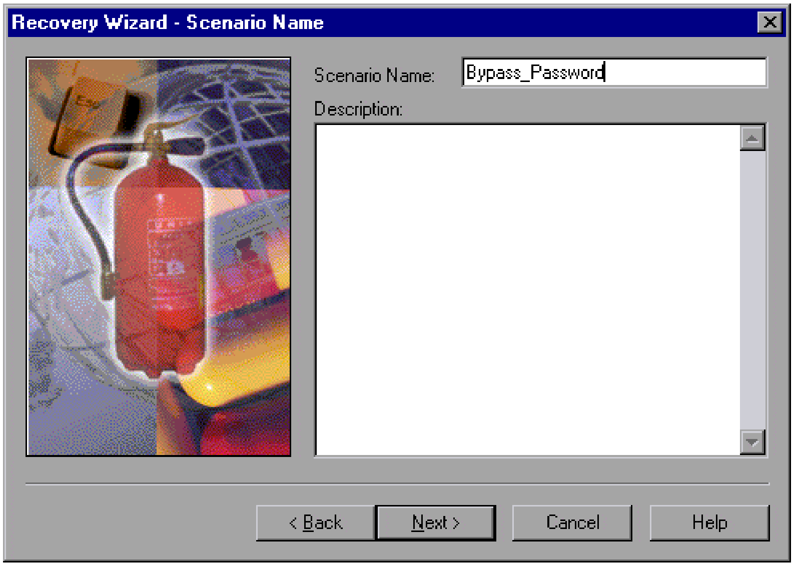
Notes:

By default, WinRunner identifies a crash event when a window opens containing the string: Application Error . You can modify the string that WinRunner uses to identify crash windows in the *excp_str.ini* file located in the <*WinRunner installation folder*>*dat* folder. For more information, see "Modifying the Crash Event Window Name," on page 514.

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

Click **Next**.

5 The Scenario Name screen opens.



Recovery Wizard - Scenario Name

Scenario Name: Bypass_Password

Description:

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

Click **Next**.

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

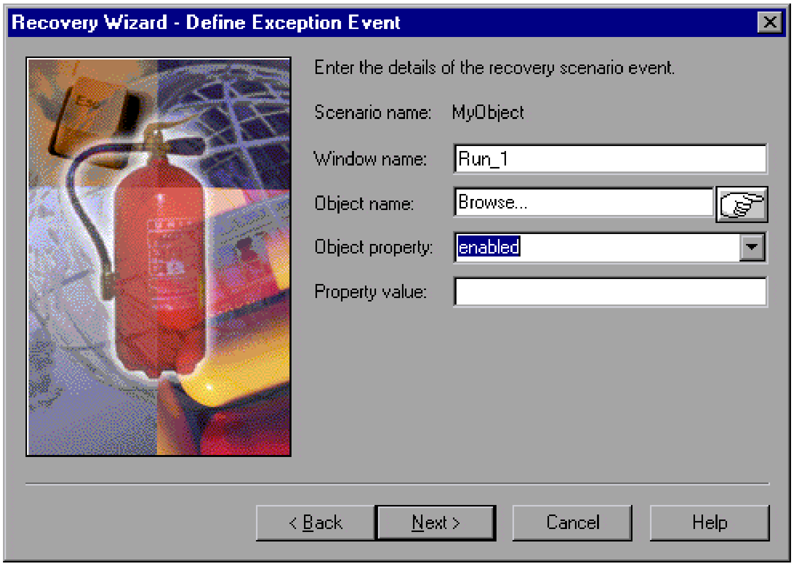
If you chose a crash event in step 4, there is no need to define the event. Proceed to step 7 on page 501.

For information on defining object events, see page 497.

For information on defining pop-up events, see page 499.

For information on defining TSL events, see page 500.

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



Recovery Wizard - Define Exception Event

Enter the details of the recovery scenario event.

Scenario name: MyObject

Window name: Run_1

Object name: Browse...

Object property: enabled

Property value:

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

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

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

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

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

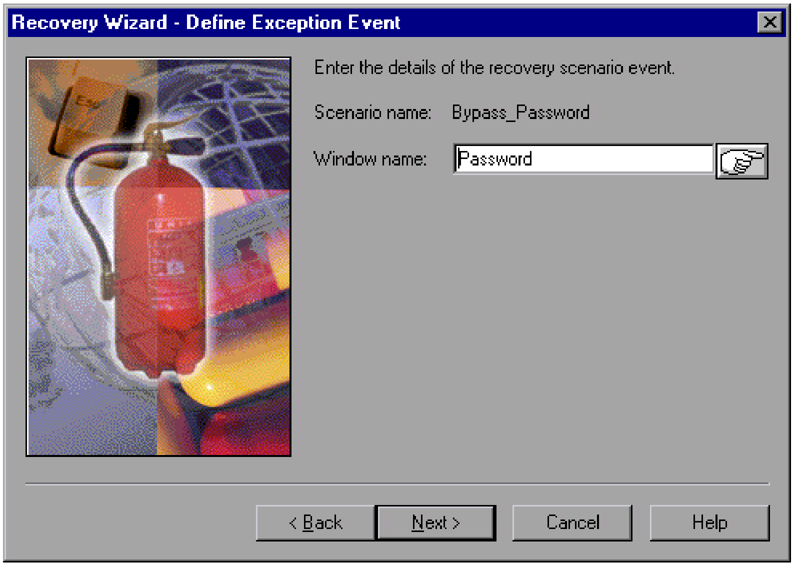
Note: You cannot specify a property that is part of the object's physical description.

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

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

Click **Next** and proceed to step 7 on page 501.

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



Recovery Wizard - Define Exception Event

Enter the details of the recovery scenario event.

Scenario name: Bypass_Password

Window name: Password

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

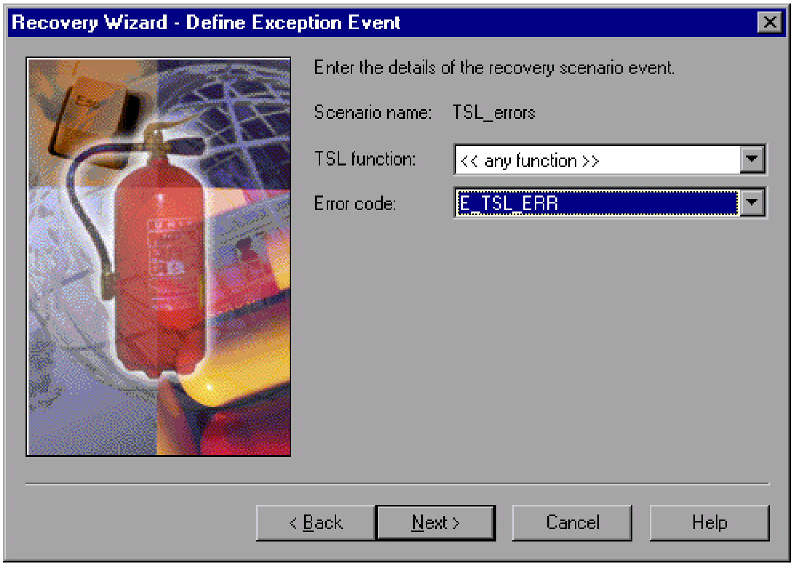
If the window is not already saved in the GUI Map and you use the pointing hand to identify the window, WinRunner automatically adds it to the active GUI Map. If the window is not already saved in the GUI Map and you type the name manually, WinRunner identifies the pop-up exception when a pop-up window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

Note: If you want to employ a **Click button** recovery operation, then the pop-up window you define must be saved in the GUI Map. If you type the window name manually, you must also add the window to the GUI Map. For more information about recovery operations, see page 501.

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

Click **Next** and proceed to step 7 on page 501.

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



Recovery Wizard - Define Exception Event

Enter the details of the recovery scenario event.

Scenario name: TSL_errors

TSL function: << any function >>

Error code: E_TSL_ERR

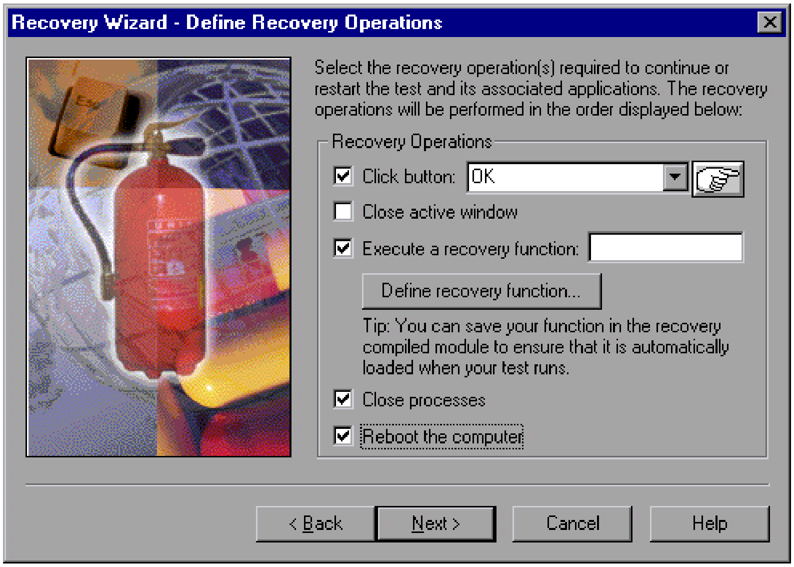
- **TSL function:** Select the TSL function for which you want to define the exception event. Select a TSL function from the list. WinRunner detects the exception only when the selected TSL function returns the code selected in the Error code box.

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

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

  Click **Next**.

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



Recovery Wizard - Define Recovery Operations

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

Recovery Operations

☑ Click button: OK

□ Close active window

☑ Execute a recovery function:

Define recovery function...

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

☑ Close processes

☑ Reboot the computer

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

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

Notes:

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

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

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

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

➤ **Execute a recovery function**: Instructs WinRunner to run the specified function when the exception event occurs. You can specify an existing function or click **Define recovery function** to define a new function. For more information on defining recovery functions, see "Defining Recovery Scenario Functions," on page 509.

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

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

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

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

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

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

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

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

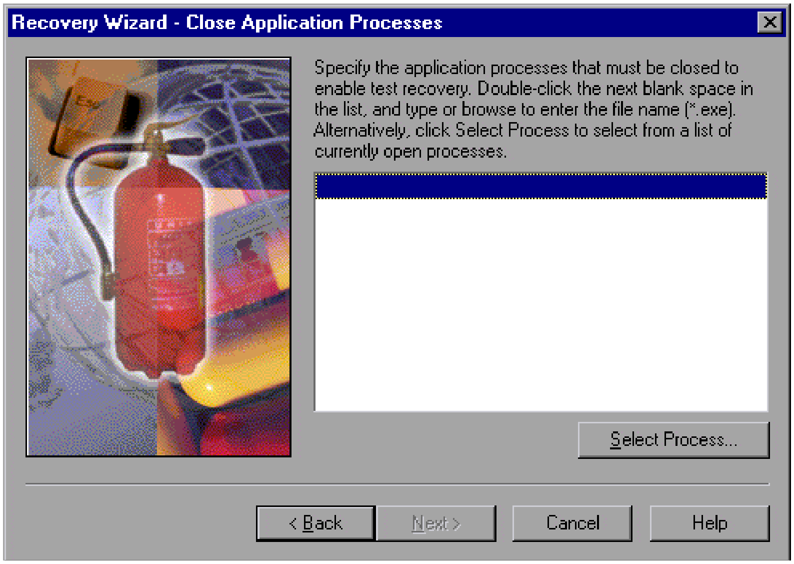
Click **Next**.

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

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

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

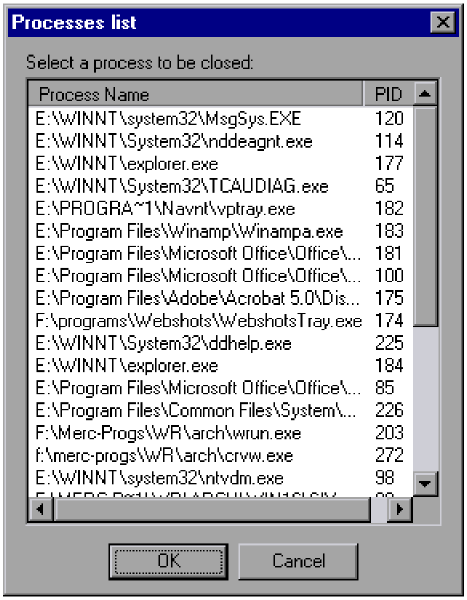
8 The Close Application Processes screen opens.



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

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

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



Processes list

Select a process to be closed:

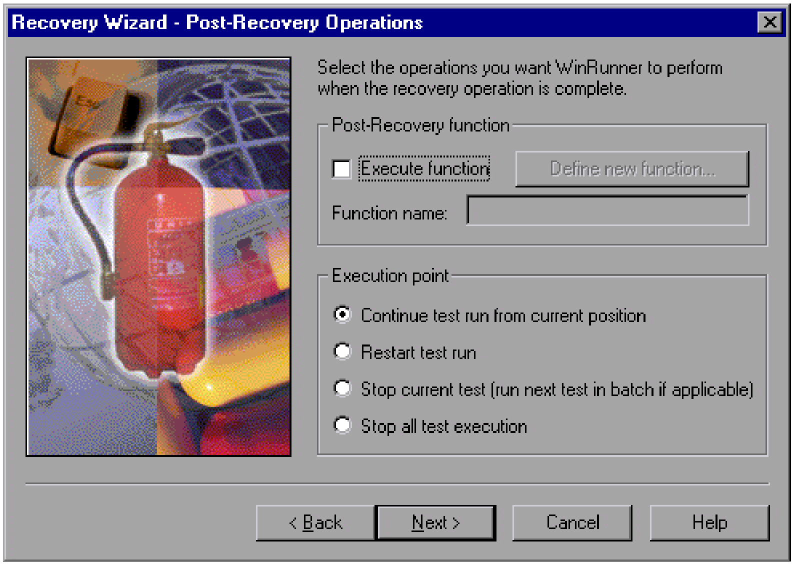
Process Name	PID
E:\WINNT\system32\MsgSys.EXE	120
E:\WINNT\System32\nddeagnt.exe	114
E:\WINNT\explorer.exe	177
E:\WINNT\System32\TCAUDIAG.exe	65
E:\PROGRA~1\Navnt\vptray.exe	182
E:\Program Files\Winamp\Winampa.exe	183
E:\Program Files\Microsoft Office\Office\...	181
E:\Program Files\Microsoft Office\Office\...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

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

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

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

9 The Post-Recovery Operations screen opens.



Recovery Wizard - Post-Recovery Operations

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

Post-Recovery function

□ Execute function

Function name:

Execution point

● Continue test run from current position

○ Restart test run

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

○ Stop all test execution

Choose from the following options:

- **Execute function:** Instructs WinRunner to run the specified function when the recovery operations are complete. You can specify an existing function or click **Define new function** to define a new function. For more information on defining post-recovery functions, see "Defining Recovery Scenario Functions," on page 509.

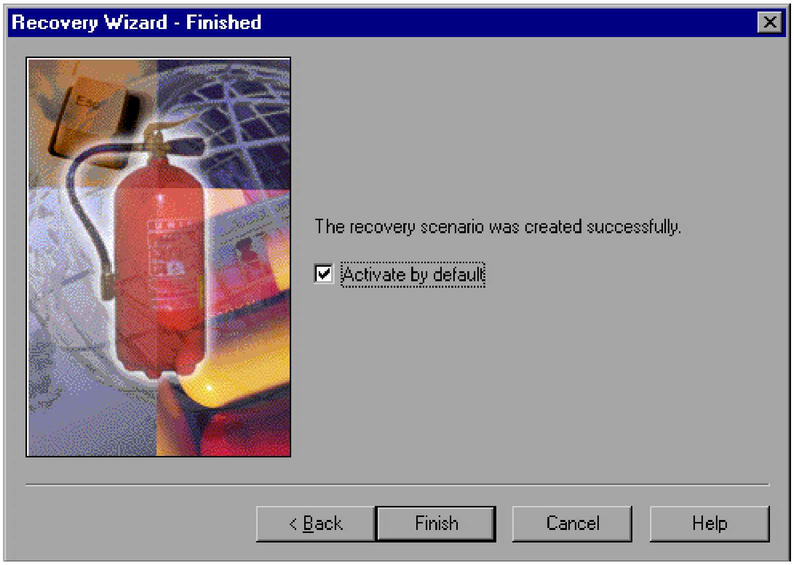
Tip: The compiled module containing the function must be loaded when the test runs. Save your function in the recovery compiled module to ensure that it is always automatically loaded when WinRunner opens. For more information on the recovery compiled module, see "Defining Recovery Scenario Functions," on page 509.

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

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

Click **Next.**

10 The Finished screen opens.



Recovery Wizard - Finished

The recovery scenario was created successfully.

☑ Activate by default

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

- Select **Activate by default** to instruct WinRunner to automatically activate the recovery scenario by default when WinRunner opens, even if the scenario was set as inactive at the end of the previous WinRunner session.
- Clear **Activate by default** if you do not want WinRunner to automatically activate the recovery scenario by default when WinRunner opens. Note that if you clear this check box, your recovery scenario will not be activated unless you activate it manually by toggling the check box in the Recovery Manager dialog box. For more information, see "Activating and Deactivating Recovery Scenarios," on page 514.

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

Defining Recovery Scenario Functions

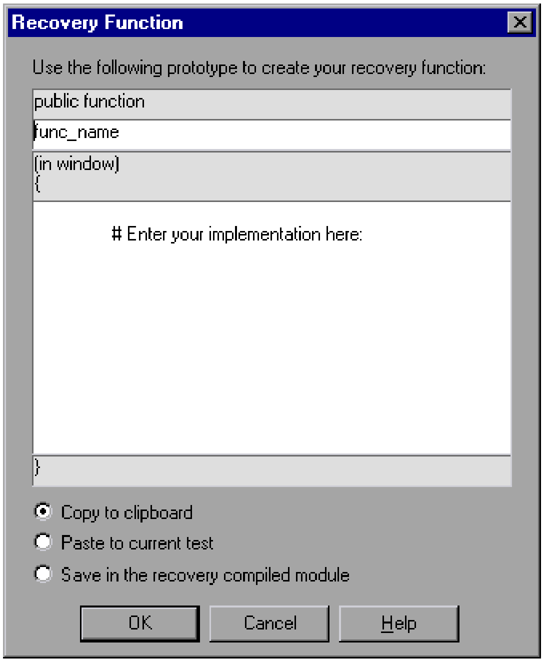
You can define recovery functions that instruct WinRunner to respond to an exception event in a way that meets your specific testing needs. You can also define post-recovery functions for compound recovery scenarios. These functions can be useful to re-open applications that may have closed when the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

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

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

To define a recovery or post-recovery function:

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



Recovery Function

Use the following prototype to create your recovery function:

```
public function
func_name
(in window)
{
        # Enter your implementation here:
}
```

● Copy to clipboard

○ Paste to current test

○ Save in the recovery compiled module

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

3 In the implementation box, enter the function content.

4 Choose how you want to store the function:

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

Notes:

If you have not defined a recovery compiled module in the **Run** > **Recovery** category of the General Options dialog box, the **Save in the recovery compiled module** option is disabled. For more information, see "Choosing the Recovery Compiled Module," on page 518.

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

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

---

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

Managing Recovery Scenarios

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

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

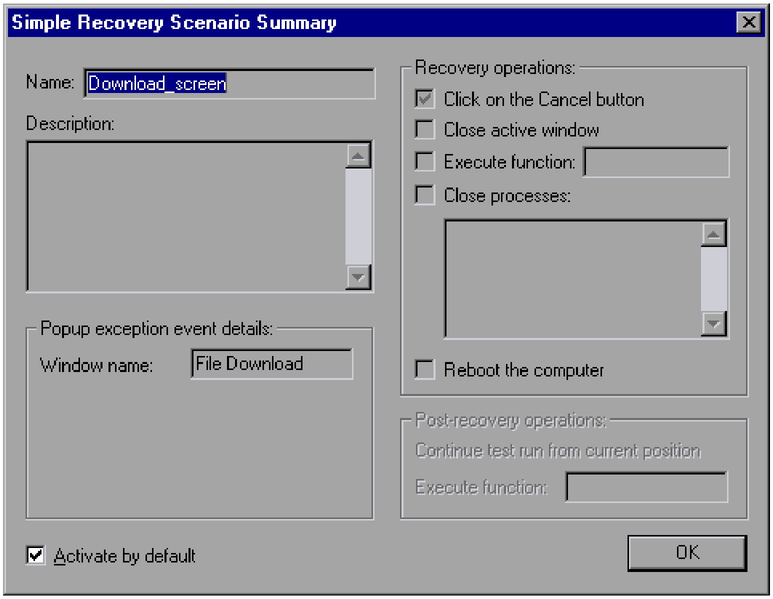
If you use crash recovery scenarios, you can also modify the string that WinRunner uses to identify crash windows.

Viewing Recovery Scenario Details

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

To open the Recovery Scenario Summary dialog box:

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



Simple Recovery Scenario Summary

Name: Download_screen

Description:

Popup exception event details:

Window name: File Download

Recovery operations:

☑ Click on the Cancel button

☐ Close active window

☐ Execute function:

☐ Close processes:

☐ Reboot the computer

Post-recovery operations:

Continue test run from current position

Execute function:

☑ Activate by default

2 Review the settings for the recovery scenario.

3 Select or clear the **Activate by default** check box if you want to modify the setting. For more information, see "Activating and Deactivating Recovery Scenarios," on page 514.

Modifying Recovery Scenarios

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

To modify a recovery scenario:

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

2 The Recovery wizard opens to the Scenario Name screen.

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

3 Navigate through the Recovery wizard and modify the details as needed. For information on the Recovery wizard options, see “Defining Simple Recovery Scenarios,” on page 483 or “Defining Compound Recovery Scenarios,” on page 494.

Deleting Recovery Scenarios

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

For more information on the recovery scenarios file, see “Setting Recovery Options,” on page 806.

To delete a recovery scenario:

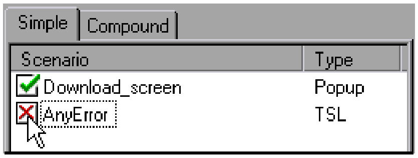
Select the recovery scenario you want to delete from the Recovery Manager dialog box and click **Delete**.

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

Activating and Deactivating Recovery Scenarios

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

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



Scenario	Type
☑ Download_screen	Popup
☒ AnyError	TSL

- Select a recovery scenario in the Recovery Manager and click **Summary** or double-click the recovery scenario to open the Recovery Scenario Summary dialog box, and select or clear the **Activate by default** check box.
- Select a recovery scenario in the Recovery Manager, click **Modify** to open the Recovery wizard, navigate to the Finished screen and select or clear the **Activate by default** check box.
- Activate a recovery scenario during the test run using TSL commands. For more information on these functions, see "Working with Recovery Scenarios in Your Test Script," on page 520.

Modifying the Crash Event Window Name

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

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

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

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

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

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

Working with Recovery Scenarios Files

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

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

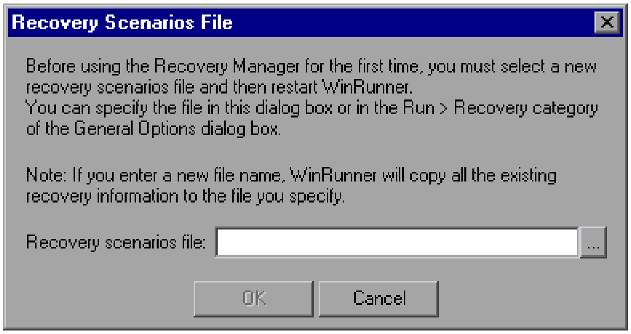
Using the Recovery Manager for the First Time

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

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

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

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



Recovery Scenarios File

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

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

Recovery scenarios file:

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

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

Choosing the Active Recovery Scenarios File

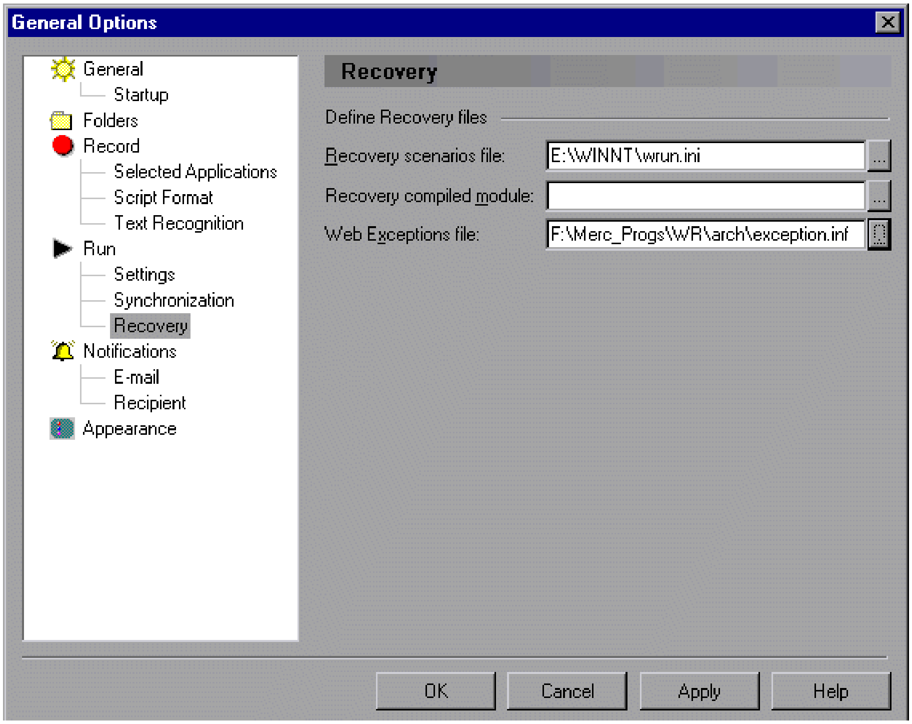
You select the active recovery scenarios file in **Run** > **Recovery** category in the General Options dialog box. You can select an existing file or enter a new file name.

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

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

To select an active recovery scenarios file

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



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

Recovery compiled module:

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

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

Choosing the Recovery Compiled Module

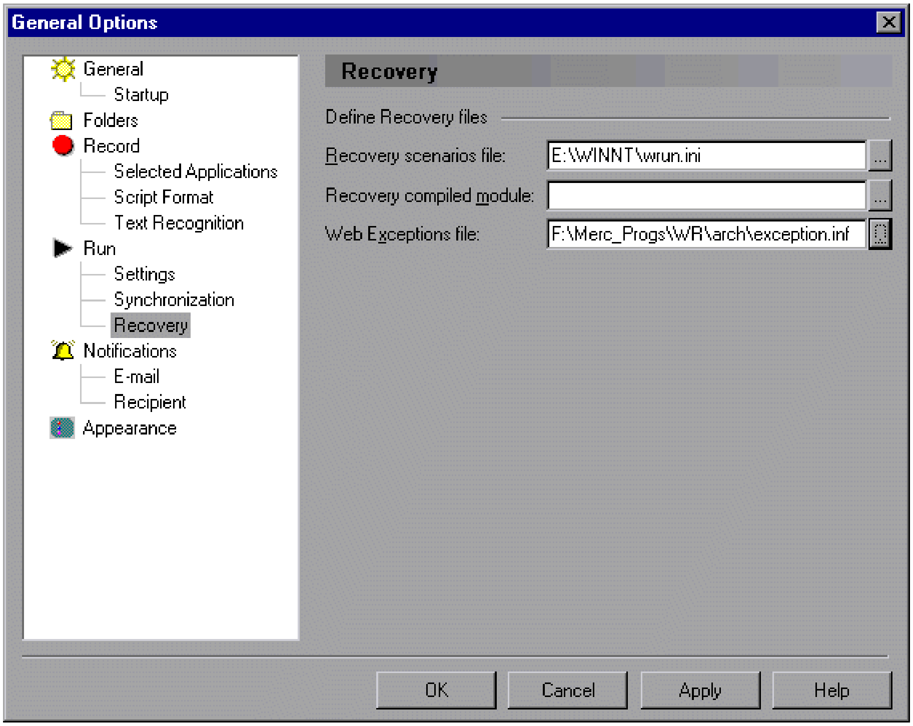
The recovery compiled module is a special compiled module that is always loaded when WinRunner opens so that the functions it contains can be accessed whenever WinRunner performs a recovery scenario.

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

To select an active recovery scenarios file

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

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



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

Recovery compiled module:

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

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

For more information on compiled modules, see Chapter 30, “Creating Compiled Modules.”

For more information on the selecting the recovery compiled module file, see “Setting Recovery Options,” on page 806.

Working with Recovery Scenarios in Your Test Script

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

Activating and Deactivating Recovery Scenarios During the Test Run

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

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

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

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

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

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

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

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

Defining Simple Recovery Scenarios Using TSL

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

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

To create compound recovery scenarios, which enable you to define crash events and/or multiple recovery operations, use the Recovery Manager. For more information, see “Defining Compound Recovery Scenarios,” on page 494.

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

24

Handling Web Exceptions

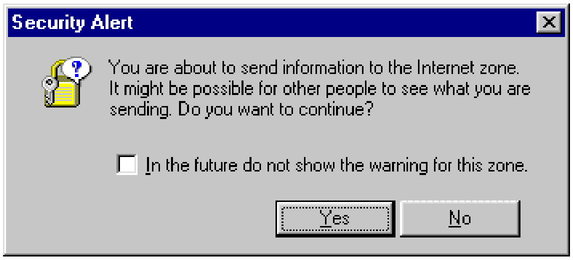
You can instruct WinRunner to handle unexpected events and errors that occur in your testing environment while testing your Web site.

This chapter describes:

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

About Handling Web Exceptions

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



Security Alert

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

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

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

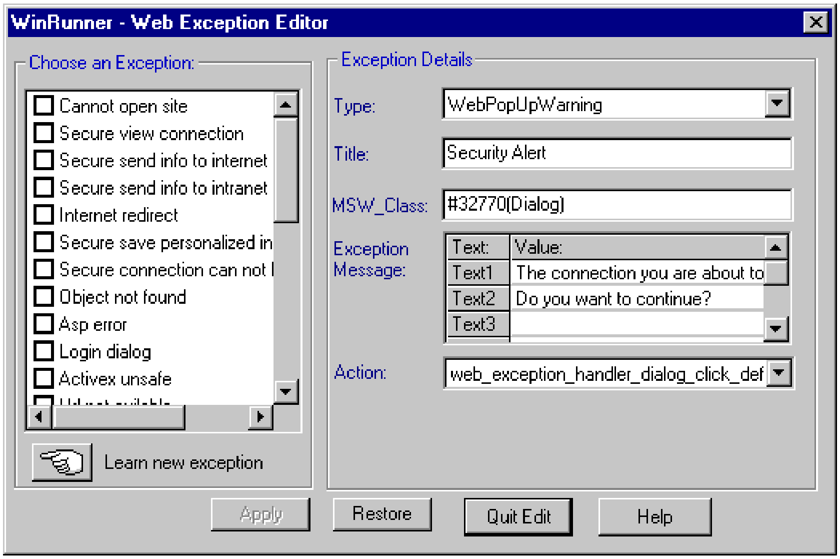
For information on loading WinRunner with the WebTest add-in, see "Loading WinRunner Add-Ins" on page 20.

Defining Web Exceptions

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

To define a Web exception:

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



Type: WebPopUpWarning

Title: Security Alert

MSW_Class: #32770(Dialog)

Exception Message:

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

Action: web_exception_handler_dialog_click_def

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

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

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

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

- **Web_exception_handler_dialog_click_default** activates the default button.
- **Web_exception_handler_fail_retry** activates the default button and reloads the Web page.
- **Web_exception_enter_username_password** uses the given user name and password.
- **Web_exception_handler_dialog_click_yes** activates the **Yes** button.
- **Web_exception_handler_dialog_click_no** activates the **No** button.

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

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

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

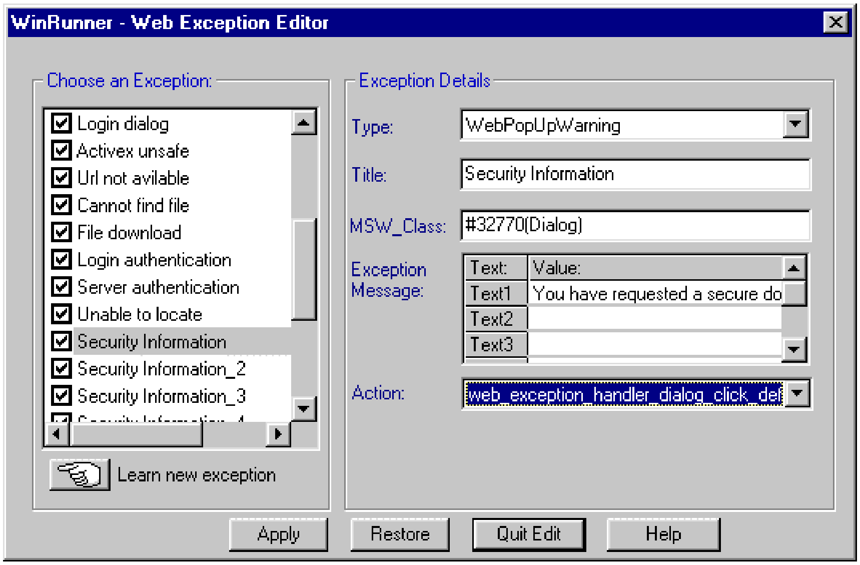
Modifying an Exception

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

To modify the details of an exception:

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

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



Choose an Exception:

- ☑ Login dialog
- ☑ Activex unsafe
- ☑ Url not avilable
- ☑ Cannot find file
- ☑ File download
- ☑ Login authentication
- ☑ Server authentication
- ☑ Unable to locate
- ☑ Security Information
- ☑ Security Information_2
- ☑ Security Information_3
- ☑ Security Information_4

Type: WebPopUpWarning

Title: Security Information

MSW_Class: #32770(Dialog)

Exception Message:

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

Action: web_exception_handler_dialog_click_def

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

3 To modify the exception category, choose a type from the **Type** list.

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

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

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

- ➤ **Web_exception_handler_dialog_click_default** activates the default button.
- ➤ **Web_exception_handler_fail_retry** activates the default button and reloads the Web page.
- ➤ **Web_exception_enter_username_password** uses the given user name and password.
- ➤ **Web_exception_handler_dialog_click_yes** activates the **Yes** button.
- ➤ **Web_exception_handler_dialog_click_no** activates the **No** button.

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

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

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

Activating and Deactivating Web Exceptions

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

To change the status of an exception:

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

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

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

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

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

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

25

Using Regular Expressions

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

- About Regular Expressions
- 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 533 for more information.

When to Use Regular Expressions

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

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

Using a Regular Expression in a GUI Checkpoint

You can use a regular expression in a GUI checkpoint, when evaluating the contents of an edit object or a static text object with a varying name. You define the regular expression by creating a GUI checkpoint on the object in which you specify the checks. The example below illustrates how to use a regular expression if you choose **Insert** > **GUI Checkpoint** > **For Object/Window** and double-click a static text object. Note that you can also use a regular expression with the **Insert** > **GUI Checkpoint** > **For Multiple Objects** command. For additional information about GUI checkpoints, see Chapter 12, "Checking GUI Objects."

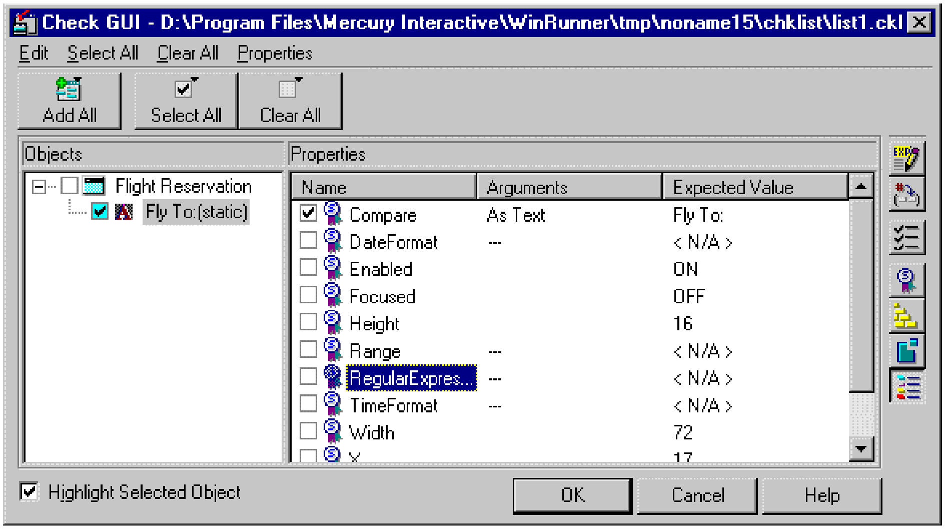
To define a regular expression in a GUI checkpoint:

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

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

2 Double-click a static text object.

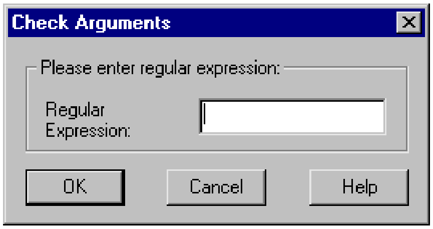
3 The Check GUI dialog box opens:



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

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

The Check Arguments dialog box opens:



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 213.

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 7, "Editing the GUI Map."

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 (^).

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

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

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

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

```
[]g-m]
```

matches the “]“ and g through m.

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

Matching Specific Characters

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

```
Q*
```

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

A period "." followed by an asterisk "*" instructs WinRunner to locate the preceding characters followed by any combination of characters. For example, in the following physical description, the regular expression enables WinRunner to locate any push button that starts with "O" (for example, On or Off):

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

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

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

Part IV

Programming with TSL

26

Enhancing Your Test Scripts with Programming

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

This chapter describes:

- About Enhancing Your Test Scripts with Programming
- Using Descriptive Programming
- Adding Comments and White Space
- Understanding Constants and Variables
- Performing Calculations
- Creating Stress Conditions
- Incorporating Decision-Making Statements
- Sending Messages to the Test Results Window
- Starting Applications from a Test Script
- Defining Test Steps
- Comparing Two Files
- Checking the Syntax of your TSL Script

Note: If you are working with WinRunner Runtime, you cannot create a test or modify a test script.

About Enhancing Your Test Scripts with Programming

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

For example:

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

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

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

TSL includes four types of functions:

- *Context Sensitive* functions perform specific tasks on GUI objects, such as clicking a button or selecting an item from a list. Function names, such as **button_press** and **list_select_item**, reflect the function's purpose.
- *Analog* functions depict mouse clicks, keyboard input, and the exact coordinates traveled by the mouse.

- *Standard* functions perform general purpose programming tasks, such as sending messages to a report or performing calculations.
- *Customization* functions allow you to adapt WinRunner to your testing environment.

WinRunner includes a visual programming tool which helps you to quickly and easily add TSL functions to your tests. For more information, see Chapter 27, "Generating Functions."

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

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

Using Descriptive Programming

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

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

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

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

Note: You can modify the logical name of any object in the GUI Map to make it easier for you to identify in your test. For more information, see "Modifying Logical Names and Physical Descriptions," on page 81.

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

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

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

The general syntax is:

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

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

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

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

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

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

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

If you are not sure which properties and values you can use to identify an object, use the GUI Spy to view the current properties and values of the object. For more information, see "Viewing GUI Object Properties," on page 34.

Adding Comments and White Space

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

Using Comments

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

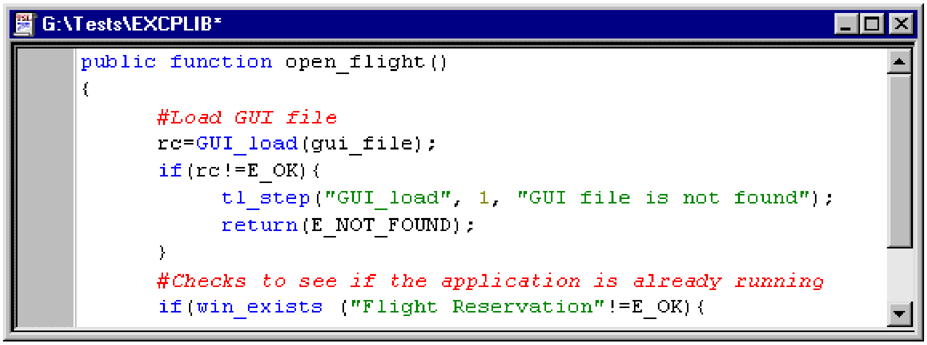
For example:

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

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

You can use the **Insert comments and indent statements** option in the **Record** > **Script Format** category of the General Options dialog box to have WinRunner automatically divide your test script into sections while you record based on window focus changes. When you choose this option, WinRunner automatically inserts a comment at the beginning of each section with the name of the window and indents the statements under each comment. For more information on the Insert comments and indent statements option, see “Setting Script Format Options” on page 786.

Inserting White Space

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



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

Understanding Constants and Variables

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

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

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

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

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

WinRunner reads the value that appears in the File Name text box and stores it in the *text* variable. A **report_msg** statement is used to display the value of the text variable in a report. For more information, see "Sending Messages to the Test Results Window" on page 552. For information about variable and constant declarations, see Chapter 29, "Creating User-Defined Functions."

Performing Calculations

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

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

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

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

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

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

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

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

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

Creating Stress Conditions

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

For Loop

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

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

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

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

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

While Loop

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

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

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

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

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

Do/While Loop

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

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

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

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

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

Incorporating Decision-Making Statements

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

If/Else Statement

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

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

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

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

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

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

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

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

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

Switch Statement

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

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

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

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

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

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

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

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

Sending Messages to the Test Results Window

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

report_msg (*message* **);**

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

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

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

Starting Applications from a Test Script

You can start an application from a WinRunner test script using the **invoke_application** function. For example, you can open the application being tested every time you start WinRunner by adding an **invoke_application** statement to a startup test. See Chapter 46, "Initializing Special Configurations," for more information on startup tests.

Tip: You can use the **Run** tab of the Test Properties dialog box to open an application at the beginning of a test run. For more information, see "Defining Startup Applications and Functions" on page 758.
You can also use a **system** statement to start an application. For more information, refer to the *WinRunner TSL Reference Guide*.

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

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

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

For example, the statement:

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

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

Defining Test Steps

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

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

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

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

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

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

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

Note that if you are using 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:\win\win.ini");
set_window("win.ini - WordPad",1);

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

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

For information on viewing the results of file comparison, see Chapter 34, “Analyzing Test Results.”

Checking the Syntax of your TSL Script

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

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

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

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

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

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

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

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

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

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

27

Generating Functions

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

This chapter describes:

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

About Generating Functions

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

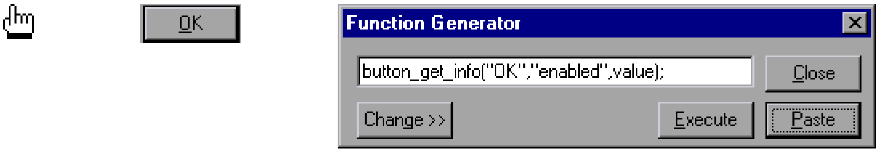
The Function Generator provides a quick, error-free way to program scripts. You can:

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

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

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

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



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

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

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

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

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

Note: You can customize the Function Generator to include the user-defined functions that you most frequently use in your test scripts. You can add new functions and new categories and sub-categories to the Function Generator. You can also set the default function for a new category. For more information, see Chapter 45, "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 566.

Generating a Function for a GUI Object

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

Using the Default Function for a GUI Object

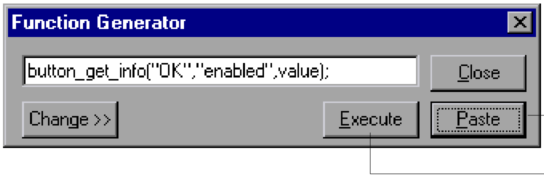
When you generate a function by pointing to a GUI object in your application, WinRunner determines the class of the object and suggests a function. For most classes, the default function is a **get** function. For example, if you click a list, WinRunner suggests the **list_get_selected** function.

To use the default function for a GUI object:

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

   To cancel the operation without selecting an object, click the right mouse button.
4. To *paste* the statement into the script, click **Paste**. The function is pasted into the test script at the insertion point and the Function Generator dialog box closes.

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



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

Pastes the function into the script

Only executes the function

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

Selecting a Non-Default Function for a GUI Object

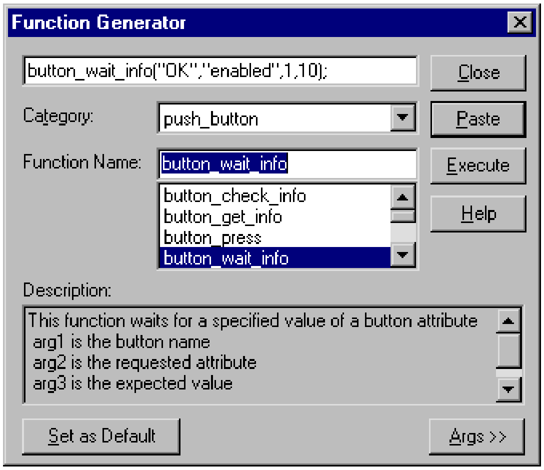
If you do not want to use the default function suggested by WinRunner, you can choose a different function from a list.

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

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

   To cancel the operation without selecting an object, click the right mouse button.
4. In the Function Generator dialog box, click **Change**. The dialog box expands and displays a list of functions. The list includes only functions that can be used on the GUI object you selected. For example, if you select a push button, the list displays **button_get_info**, **button_press**, etc.

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



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

Category: push_button

Function Name: button_wait_info

button_check_info
button_get_info
button_press
button_wait_info

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

6 If you want to modify the argument values, click **Args**. The dialog box expands and displays a text box for each argument. See "Assigning Argument Values" on page 564 for more information on how to fill in the argument text boxes.

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

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

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

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

Selecting a Function from a List

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

To select a function from a list:

1. Choose **Insert** > **Function** > **From Function Generator** or click the **Insert Function from Function Generator** button on the User toolbar to open the Function Generator dialog box.
2. In the **Category** list, select a function category. For example, if you want to view menu functions, select menu. If you do not know which category you need, use the default *all_functions*, which displays all the functions listed in alphabetical order.
3. In the **Function Name** list, choose a function. If you select a category, only the functions that belong to the category are displayed in the list. The generated statement appears at the top of the dialog box. Note that WinRunner automatically fills in the default argument values. A description of the function appears at the bottom of the dialog box.
4. To define or modify the argument values, click **Args**. The dialog box expands and displays a text box for each argument. See "Assigning Argument Values" on page 564 to learn how to fill in the argument text boxes.
5. To *paste* the statement into the test script, click **Paste**. The function is pasted into the test script at the insertion point.

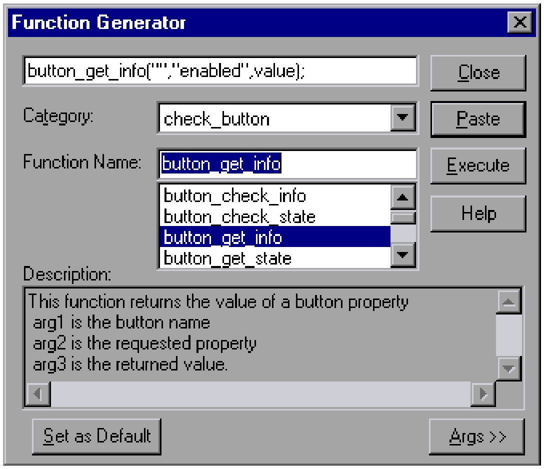
   To *execute* the function, click **Execute**. The function is immediately executed but is not pasted into the test script.
6. You can continue to generate additional function statements by repeating the steps above without closing the dialog box.
7. Click **Close** to close the dialog box.

Assigning Argument Values

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

To assign or modify argument values for a generated function:

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



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

Category: check_button

Function Name: button_get_info

button_check_info
button_check_state
button_get_info
button_get_state

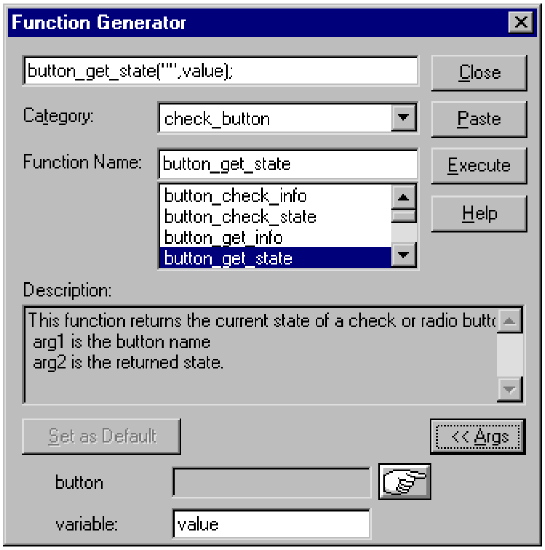
Description:

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

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

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

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



button_get_state("",value);

Category: check_button

Function Name: button_get_state

button_check_info
button_check_state
button_get_info
button_get_state

Description:

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

button:

variable: value

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

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

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

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

Modifying the Default Function in a Category

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

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

To change the default function in a category:

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

The selected function remains the default function in its category until it is changed or until you end your WinRunner session. To permanently save changes to the default function setting, add a **generator_set_default_function** statement to your startup test. For more information on startup tests, see Chapter 46, "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.

28

Calling Tests

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

This chapter describes:

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

About Calling Tests

By adding **call** statements to test scripts, you can create a modular test tree structure containing an entire test suite. A modular test tree consists of a main test that calls other tests, 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 35, "Running Batch Tests."

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

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

You can also use the Insert Call to QuickTest dialog box or insert a **call_ex** statement to call a QuickTest test. For more information, see Chapter 47, "Integrating with QuickTest Professional."

Using the Call Statement

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

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

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

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

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

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

The *test_name* is the name of the test to invoke. The *parameters* are the 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. For more information, see “Viewing the Call Chain” on page 579.

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.

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");
...
```

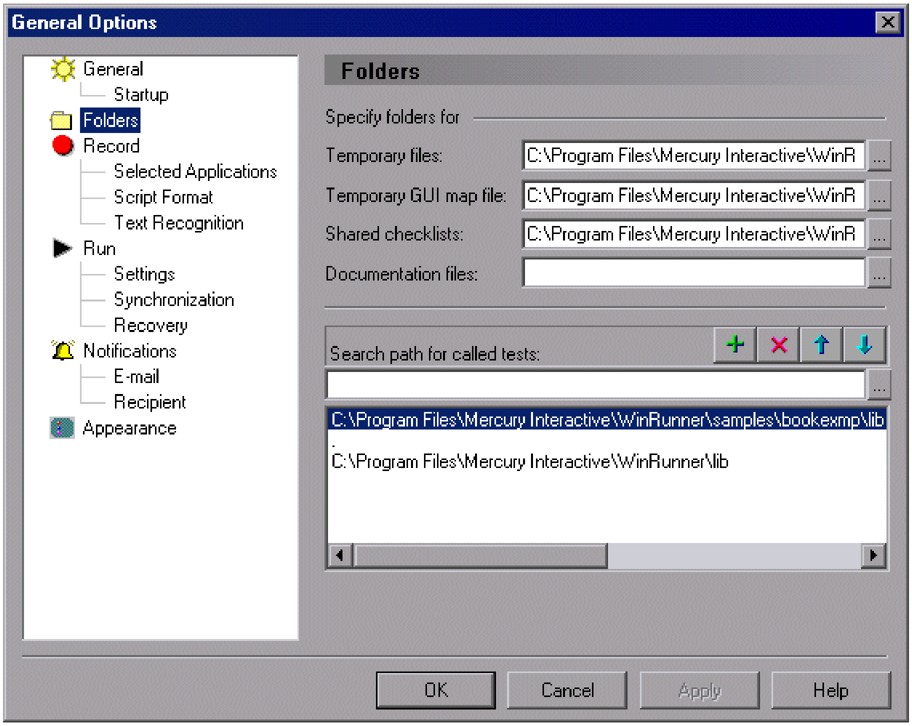
In the above example, batch_test calls help_test. In help_test, if a particular message appears on the screen, execution is stopped and control is returned to the batch test. Note that the return value of the help_test is also returned to the batch test, and is assigned to the variable *return_val*.

For more information on batch tests, see Chapter 35, “Running Batch Tests.”

Setting the Search Path

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

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



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

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

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

Documentation files:

Search path for called tests:

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

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

For more information about how to set a search path in the General Options dialog box, see Chapter 41, "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 all tests in the current session only.

For example:

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

This statement tells WinRunner to search the *c:\ui_tests* folder for called tests. For more information on using the **setvar** function, see Chapter 44, "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 934.

Replacing Data from the Test with Parameters

If you pass parameters to a called test using **call** or **call_close**, the parameters must be declared in the called test. Otherwise, a warning message will be displayed ("Warning: Test <path to test>: too many arguments").

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.

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

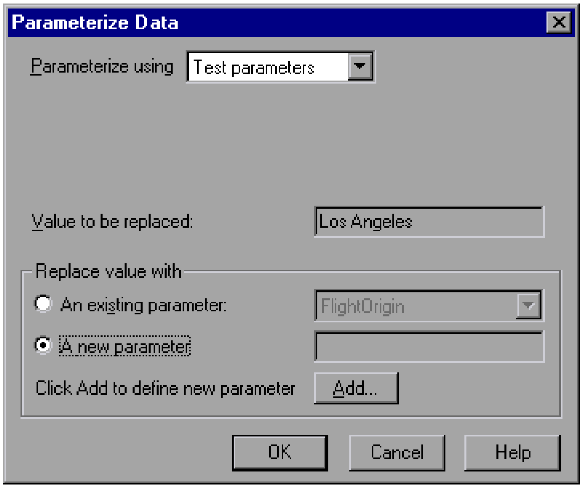
- Use the **Parameters** tab of the Test Properties dialog box when you want to manage the parameters of the test including adding, modifying, and deleting the parameters list for the test. For more information about **Parameters** tab of the Test Properties dialog box, see "Managing Test Parameters" on page 753.
- Use the Parameterize Data dialog box when you want to replace data from the test with existing parameters. You can also create new parameters from this dialog box.

Defining Test Parameters in the Parameterize Data Dialog Box

You can replace a selected value in your test script with an existing or new parameter using the Parameterize Data dialog box.

To parameterize statements using test parameters:

1. In your test script, select the first instance in which you have data that you want to parameterize.
2. Choose **Table** > **Parameterize Data**. The Parameterize Data dialog box opens.
3. In the **Parameterize using** box, select **Test parameters**.



Parameterize Data

Parameterize using: Test parameters

Value to be replaced: Los Angeles

Replace value with

○ An existing parameter: FlightOrigin

● A new parameter

Click Add to define new parameter

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

➤ If you select **An existing parameter**, select the parameter you want to use. Note that the parameters listed here are the same as those listed in the **Parameters** tab of the Test Properties dialog box.

➤ If you select **A new parameter**, click the **Add** button. The Parameter Properties dialog box opens. Add a new parameter as described on page 753. The new parameter is displayed in the new parameter field. The new parameter is also added to the **Parameters** tab of the Test Properties dialog box.

5 Click **OK**.

The data selected in the test script is replaced with the parameter you created or selected. When the test is called by the calling test, the parameter is replaced by the value defined in the calling test.

6 Repeat steps 1 to 5 for each argument you want to parameterize.

Test Parameter Scope

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

Parameters that are expressions, variables, or array elements are evaluated and then passed to the called test. This means that a copy is passed to the called test. This copy is local; if its value is changed in the called test, the original value in the calling test is not affected. For example:

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

In the calling test (test_1), the variable *i* is assigned the value 5. This value is passed to the called test (test_2) as the value for the formal parameter x.

Note that when a new value (8) is assigned to x in test_2, this change does not affect the value of *i* in test_1.

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

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

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

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

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

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

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

Viewing the Call Chain

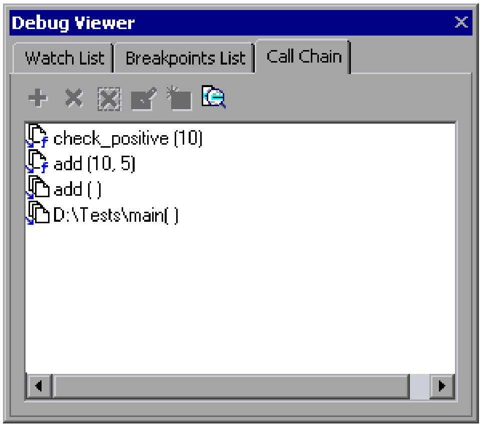
At each break during a test run—such as after a Step command, at a breakpoint, or at the end of a test, you can view the current chain of called tests and functions in the Call Chain pane of the Debug Viewer window.

To view the current call chain:

1. If the Debug Viewer window is not currently displayed, or the Call Chain pane is not open in the window, choose **Debug** > **Call Chain** to display it. If the Call Chain pane is open, but a different pane is currently displayed, click the **Call Chain** tab to display it.
2. Ensure that your called tests have breakpoints in places where you would like to view the call chain. Alternatively, use the Step commands to control the run of the test.

   For more information on Step commands, see Chapter 37, "Controlling Your Test Run."

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



```
check_positive (10)
add (10, 5)
add ( )
D:\Tests\main( )
```

Tip: The Debug Viewer window can be displayed as a docked window within the WinRunner window, or it can be a floating window that you can drag to any location on your screen. By default the Debug Viewer opens as a docked window on the right side of the WinRunner screen. To move the window to another location, drag the Debug Viewer titlebar.

---

4 To view the script of a test in the call chain, double-click a test or function, or select the test or function in the list and click **Display test**. The selected test or function becomes the active window in WinRunner.

29

Creating User-Defined Functions

You can expand WinRunner's testing capabilities by creating your own TSL functions. You can use these user-defined functions in a test or a compiled module. This chapter describes:

- About Creating User-Defined Functions
- Function Syntax
- Return and Exit Statements
- Variable, Constant, and Array Declarations
- Example of a User-Defined Function

About Creating User-Defined Functions

In addition to providing built-in functions, TSL allows you to design and implement your own functions. 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 30, "Creating Compiled Modules."
- Call functions from the Microsoft Windows API or any other external functions stored in a DLL. For more information, see Chapter 31, "Calling Functions from External Libraries."

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

For example, you could create a function called open_flight that loads a GUI map file, starts the Flight Reservation application, and logs into the system, or resets the main window if the application is already open.

A function can be called from anywhere in a test script. Since it is already compiled, execution time is accelerated. For instance, suppose you create a test that opens a number of files and checks their contents. Instead of recording or programming the sequence that opens the file several times, you can write a function and call it each time you want to open a file.

Function Syntax

A user-defined function has the following structure:

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

Class

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

Once you execute a public function, it is available to all tests, for as long as the test containing the function remains open. This is convenient when you want the function to be accessible from called tests. However, if you want to create a function that will be available to many tests, you should place it in a compiled module. The functions in a compiled module are available for the duration of the testing session.

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

Parameters

Parameters need not be explicitly declared. They can be of mode *in*, *out*, or *inout*. For all non-array parameters, the default mode is *in*. For array parameters, the default is *inout*. The significance of each of these parameter types is as follows:

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

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

inout*:* A parameter that can be assigned a value from outside or inside the function.

A parameter designated as out or inout must be a variable name, not an expression. When you call a function containing an *out* or an *inout* parameter, the argument corresponding to that parameter must be a variable, and not an expression. For example, consider a user-defined function with the following syntax:

```
function get_date (out todays_date) { ... }
```

Proper usage of the function call would be:

```
get_date (todays_date);
```

Illegal usage of the function call would be:

```
get_date (date[i]); or get_date ("Today’s date is"& todays_date);
```

because both contain expressions.

Array parameters are designated by square brackets. For example, the following parameter list in a user-defined function indicates that variable *a* is an array:

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

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

Note: You can define up to 15 parameters in a user-defined function.

Return and Exit Statements

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

return ([*expression*] **);**

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

The texit statement can be used to stop a function or test run. The syntax is:

texit ([*expression*]);

When a test is run interactively, texit discontinues the test run entirely. When a test is run in batch mode, the statement ends execution of the current main test only; control is then returned to the calling batch test. The texit function also returns the value of the evaluated expression to the calling function or test.

Note: QuickTest does not support **texit** statements inside called functions. If QuickTest calls a WinRunner function containing a **texit** statement, the function call fails.

Variable, Constant, and Array Declarations

Declaration is usually optional in TSL. In functions, however, variables, constants, and arrays must all be declared. The declaration can be within the function itself, or anywhere else within the test script or compiled module containing the function. You can find additional information about declarations in the *TSL Reference*.

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 30, “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);
```

30

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:

- About Creating Compiled Modules
- 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.

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 46, "Initializing Special Configurations," for more information.

Notes:

You do not need to add **load** statements to your startup test or to any other test in order to load the *recovery compiled module*. The recovery compiled module is automatically loaded when you start WinRunner. For more information on the recovery compiled module, see Chapter 23, "Defining and Using Recovery Scenarios."

If you are working in the *GUI Map File per Test* mode, compiled modules do not load GUI map files. If your compiled module references GUI objects, then those objects must also be referenced in the test that loads the compiled module. For additional information, see Chapter 6, "Working in the GUI Map File per Test Mode."

Contents of a Compiled Module

A compiled module, like a regular test you create in TSL, can be opened, edited, and saved. You indicate that a test is a compiled module in the **General** tab of the Test Properties dialog box, by selecting **Compiled Module** in the Test Type box. For more information, see "Creating a Compiled Module" on page 594.

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 29, "Creating User-Defined Functions."
- prototypes of external functions. For more information, see Chapter 31, "Calling Functions from External Libraries."
- **load** statements to other modules. For more information, see "Loading and Unloading a Compiled Module" on page 595.

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

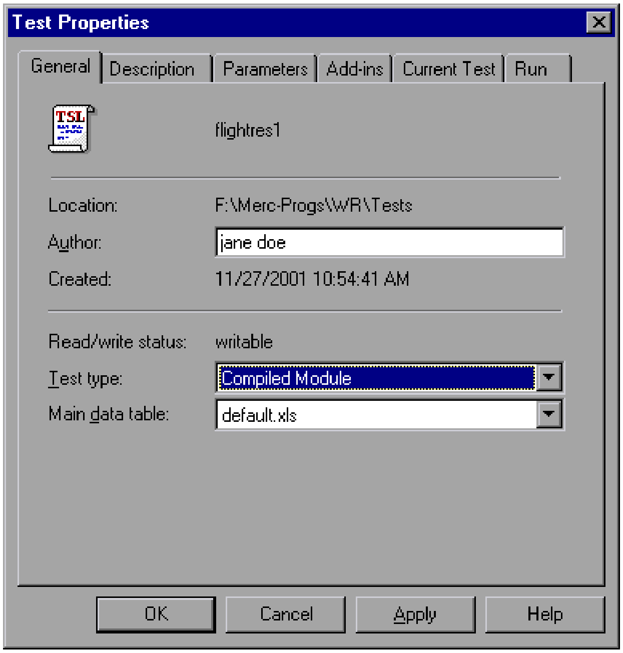
- A public function is available to all modules and tests, while a static function is available only to the module within which it was defined.
- The loaded module remains resident in memory even when test execution is aborted. However, all variables defined within the module (whether static or public)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

Creating a Compiled Module

Creating a compiled module is similar to creating a regular test script.

To create a compiled module:

1. Choose **File** > **Open** to open a new test.
2. Write the user-defined functions.
3. Choose **File** > **Test Properties** and click the **General** tab.
4. In the **Test Type** list, choose **Compiled Module** and then click **OK**.



flightres1

Location: F:\Merc-Progs\WR\Tests

Author: jane doe

Created: 11/27/2001 10:54:41 AM

Read/write status: writable

Test type: Compiled Module

Main data table: default.xls

5. Choose **File** > **Save**.

   Save your 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 574.

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 46, "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)

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;
}
```

31

Calling Functions from External Libraries

WinRunner enables you to call functions from the Windows API and from any external DLL (Dynamic Link Library).

This chapter describes:

- About Calling Functions from External Libraries
- Dynamically Loading External Libraries
- Declaring External Functions in TSL
- Windows API Examples

About Calling Functions from External Libraries

You can extend the power of your automated tests by calling functions from the Windows API or from any external DLL. For example, using functions in the Windows API you can:

- Use a standard Windows message box in a test with the *MessageBox* function.
- Send a WM (Windows Message) message to the application being tested with the *SendMessage* function.
- Retrieve information about your application's windows with the *GetWindow* function.
- Integrate the system beep into tests with the *MessageBeep* function.
- Run any windows program using *ShellExecute*, and define additional parameters such as the working directory and the window size.

- Check the text color in a field in the application being tested with *GetTextColor*. This can be important when the text color indicates operation status.
- Access the Windows clipboard using the *GetClipboard* functions.

You can call any function exported from a DLL with the _ _ stdcall calling convention. You can also load DLLs that are part of the application being tested in order to access its exported functions.

Using the **load_dll** function, you dynamically load the libraries containing the functions you need. Before you actually call the function, you must write an *extern* declaration so that the interpreter knows that the function resides in an external library.

Note: For information about specific Windows API functions, refer to 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.

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 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 46, "Initializing Special Configurations.")

The syntax of the extern declaration is:

extern *type function_name* **(** *parameter1*, *parameter2*,... **);**

The *type* refers to the return value of the function. The type can be one of the following:

char (signed and unsigned)	*float*
short (signed and unsigned)	*double*
int (signed and unsigned)	*string* (equivalent to C char*)

Each *parameter* must include the following information:

[mode] *type* [name] [<*size*>]

The *mode* can be either *in*, *out*, or *inout*. The default is *in*. Note that these values must appear in lowercase letters.

The *type* can be any of the values listed above.

An optional *name* can be assigned to the parameter to improve readability.

The <*size*> is required only for an *out* or *inout* parameter of type *string* (see below).

For example, suppose you want to call a function called set_clock that sets the time on a clock application. The function is part of an external DLL that you loaded with the **load_dll** statement. To declare the function, write:

```
extern int set_clock (string name, int time);
```

The set_clock function accepts two parameters. Since they are both input parameters, no mode is specified. The first parameter, a string, is the name of the clock window. The second parameter specifies the time to be set on the clock. The function returns an integer that indicates whether the operation succeeded.

Once the extern declaration is interpreted, you can call the set_clock function the same way you call a TSL function:

```
result = set_clock ("clock v. 3.0", 3);
```

If an extern declaration includes an *out* or *inout* parameter of type *string*, you must budget the maximum possible string size by specifying an integer <*size*> after the parameter *type* or (optional) *name*. For example, the statement below declares the function get_clock_string, that returns the time displayed in a clock application as a string value in the format "The time is...".

```
extern int get_clock_string (string clock, out string time <20>);
```

The *size* should be large enough to avoid an overflow. If no value is specified for *size*, the default is 100.

TSL identifies the function in your code by its name only. You must pass the correct parameter information from TSL to the function. TSL does not check parameters. If the information is incorrect, the operation fails.

Note: If you want to return a string value from a function in an external DLL, it is recommended to use an output parameter rather than a return value.

For example your DLL should look something like:

```
int foo(char* szRetString)
{
  ...
  strcpy(szRetString, "hi");
  return nErrCode;
}
```

And the corresponding **extern** statement should be something like:

```
extern int foo(out string);
```

In addition, your external function must adhere to the following conventions:

- Any parameter designated as a *string* in TSL must correspond to a parameter of type *char**.
- Any parameter of mode *out* or *inout* in TSL must correspond to a pointer in your exported function. For instance, a parameter *out int* in TSL must correspond to a parameter *int** in the exported function.
- The external function must observe the standard Pascal calling convention *export far Pascal*.

For example, the following declaration in TSL:

```
extern int set_clock (string name, inout int time);
```

must appear as follows in your external function:

```
int set_clock(
        char* name,
        int* time
        );
```

Windows API Examples

The following sample tests call functions in the Windows API.

Checking Window Mnemonics

This test integrates the API function *GetWindowTextA* into a TSL function that checks for mnemonics (underlined letters used for keyboard shortcuts) in object labels. The TSL function receives one parameter: the logical name of an object. If a mnemonic is not found in an object's label, a message is sent to a report.

```
# load the appropriate DLL (from Windows folder)
load ("win32api");

# define the user-defined function "check_labels"
public function check_labels(in obj)
{
    auto hWnd,title,pos,win;
    win = GUI_get_window();
    obj_get_info(obj,"handle",hWnd);
    GetWindowTextA(hWnd,title,128);
     pos = index(title,"&");
    if (pos == 0)
        report_msg("No mnemonic for object: "& obj & "in window: "& win);
}

# start Notepad application
invoke_application("notepad.exe","","",SW_SHOW);
```

```
# open Find window
set_window ("Notepad");
menu_select_item ("Search;Find...");

# check mnemonics in "Up" radio button and "Cancel" pushbutton
set_window ("Find");
check_labels ("Up");
check_labels ("Cancel");
```

Loading a DLL and External Function

This test fragment uses crk_w.dll to prevent recording on a debugging application. To do so, it calls the external *set_debugger_pid* function.

```
# load the appropriate DLL
load_dll("crk_w.dll");

# declare function
extern int set_debugger_pid(long);

# load Systems DLLs (from Windows folder)
load ("win32api");

# find debugger process ID
win_get_info("Debugger","handle",hwnd);
GetWindowThreadProcessId(hwnd,Proc);

# notify WinRunner of the debugger process ID
set_debugger_pid(Proc);
```

32

Creating Dialog Boxes for Interactive Input

WinRunner enables you to create dialog boxes that you can use to pass input to your test during an interactive test run.

This chapter describes:

- About Creating Dialog Boxes for Interactive Input
- 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

About Creating Dialog Boxes for Interactive Input

You can create dialog boxes that pop up during an interactive test run, prompting the user to perform an action—such as typing in text or selecting an item from a list. This is useful when the user must make a decision based on the behavior of the application under test (AUT) during runtime, and then enter input accordingly. For example, you can instruct WinRunner to execute a particular group of tests according to the user name that is typed into the dialog box.

To create the dialog box, you enter a TSL statement in the appropriate location in your test script. During an interactive test run, the dialog box opens when the statement is executed. By using control flow statements, you can determine how WinRunner responds to the user input in each case.

There are five different types of dialog boxes that you can create using the following TSL functions:

- **create_input_dialog** creates a dialog box with any message you specify, and an edit field. The function returns a string containing whatever you type into the edit field, during an interactive run.
- **create_list_dialog** creates a dialog box with a list of items, and your message. The function returns a string containing the item that you select during an interactive run.
- **create_custom_dialog** creates a dialog box with edit fields, check boxes, an "execute" button, and a Cancel button. When the "execute" button is clicked, the **create_custom_dialog** function executes a specified function.
- **create_browse_file_dialog** displays a browse dialog box from which the user selects a file. During an interactive run, the function returns a string containing the name of the selected file.
- **create_password_dialog** creates a dialog box with two edit fields, one for login name input, and one for password input. You use a password dialog box to limit user access to tests or parts of tests.

Each dialog box opens when the statement that creates it is executed during a test run, and closes when one of the buttons inside it is clicked.

Creating an Input Dialog Box

An input dialog box contains a custom one-line message, an edit field, and OK and Cancel buttons. The text that the user types into the edit field during a test run is returned as a string.

You use the TSL function **create_input_dialog** to create an input dialog box. This function has the following syntax:

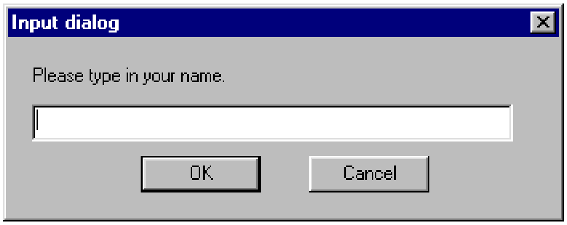
create_input_dialog (*message* **);**

The *message* can be any expression. The text appears as a single line in the dialog box.

For example, you could create an input dialog box that asks for a user name. This name is returned to a variable and is used in an **if** statement in order to call a specific test suite for any of several users.

To create such a dialog box, you would program the following statement:

```
name = create_input_dialog ("Please type in your name.");
```



The input that is typed into the dialog box during a test run is passed to the variable *name* when the **OK** button is clicked. If the **Cancel** button is clicked, an empty string (empty quotation marks) is passed to the variable *name*.

Tip: You can use the following statements to display the message that the user types in the dialog box:

```
rc=create_input_dialog("Message");
pause(rc);
```

For additional information on the **pause** function, refer to the *TSL Reference*.

Note that you can precede the message parameter with an exclamation mark. When the user types input into the edit field, each character entered is represented by an asterisk. Use an exclamation mark to prevent others from seeing confidential information.

Creating a List Dialog Box

A list dialog box has a title and a list of items that can be selected. The item selected by the user from the list is passed as a string to a variable.

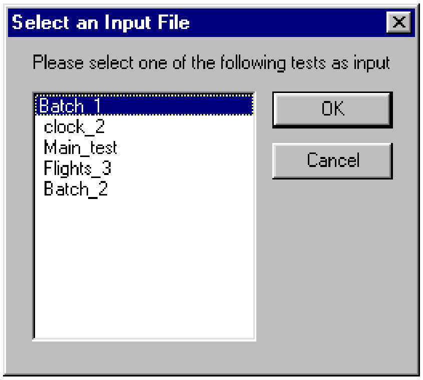
You use the TSL function **create_list_dialog** to create a list dialog box. This function has the following syntax:

create_list_dialog (*title*, *message*, *list_items* **);**

- *title* is an expression that appears in the window banner of the dialog box.
- *message* is one line of text that appear in the dialog box.
- *list_items* contains the options that appear in the dialog box. Items are separated by commas, and the entire list is considered a single string.

For example, you can create a dialog box that allows the user to select a test to open. To do so, you could enter the following statement:

```
filename = create_list_dialog ("Select an Input File", "Please select one of the
following tests as input", "Batch_1, clock_2, Main_test, Flights_3, Batch_2");
```



Select an Input File

Please select one of the following tests as input

Batch_1
clock_2
Main_test
Flights_3
Batch_2

The item that is selected from the list during a test run is passed to the variable *filename* when the **OK** button is clicked. If the **Cancel** button is clicked, an empty string (empty quotation marks) is passed to the variable *filename*.

Creating a Custom Dialog Box

A custom dialog box has a custom title, up to ten edit fields, up to ten check boxes, an "execute" button, and a Cancel button. You specify the label for the "execute" button. When you click the "execute" button, a specified function is executed. The function can be either a TSL function or a user-defined function.

You use the TSL function **create_custom_dialog** to create a custom dialog box. This function has the following syntax:

create_custom_dialog (*function_name*, *title*, *button_name*, *edit_name*$_{1\text{-}n}$, *check_name*$_{1\text{-}m}$ **);**

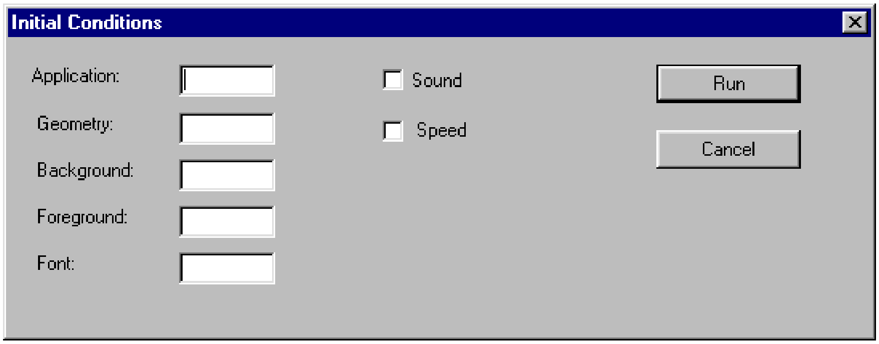
- *function_name* is the name of the function that is executed when you click the "execute" button.
- *title* is an expression that appears in the title bar of the dialog box.
- *button_name* is the label that will appear on the "execute" button. You click this button to execute the contained function.
- *edit_name* contains the labels of the edit field(s) of the dialog box. Multiple edit field labels are separated by commas, and all the labels together are considered a single string. If the dialog box has no edit fields, this parameter must be an empty string (empty quotation marks).
- *check_name* contains the labels of the check boxes in the dialog box. Multiple check box labels are separated by commas, and all the labels together are considered a single string. If the dialog box has no check boxes, this parameter must be an empty string (empty quotation marks).

When the "execute" button is clicked, the values that the user enters are passed as parameters to the specified function, in the following order:

edit_name$_1$,... *edit_name*$_n$,*check_name*$_1$,... *check_name*$_m$

In the following example, the custom dialog box allows the user to specify startup parameters for an application. When the user clicks the **Run** button, the user-defined function, run_application1, invokes the specified Windows application with the initial conditions that the user supplied.

```
res = create_custom_dialog ("run_application1", "Initial Conditions", "Run",
    "Application:, Geometry:, Background:, Foreground:, Font:", "Sound,
    Speed");
```



If the specified function returns a value, this value is passed to the variable *res*. If the **Cancel** button is clicked, an empty string (empty quotation marks) is passed to the variable *res*.

Note that you can precede any edit field label with an exclamation mark. When the user types input into the edit field, each character entered is represented by an asterisk. You use an exclamation mark to prevent others from seeing confidential information, such as a password.

Creating a Browse Dialog Box

A browse dialog box allows you to select a file from a list of files, and returns the name of the selected file as a string.

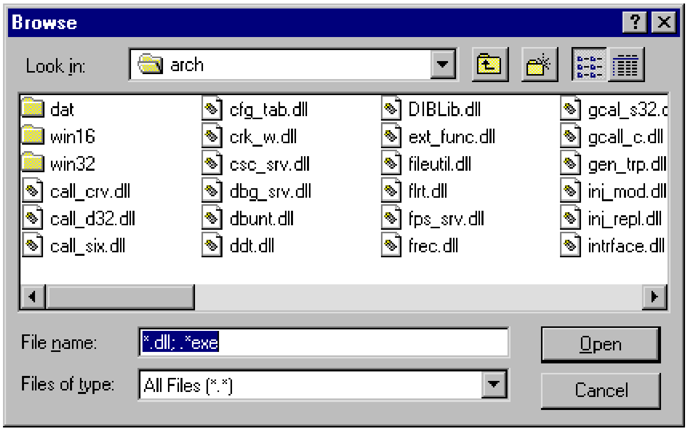
You use the TSL function **create_browse_file_dialog** to create a browse dialog box. This function has the following syntax:

create_browse_file_dialog (*filter* **);**

where *filter* sets a filter for the files to display in the Browse dialog box. You can use wildcards to display all files (*.*) or only selected files (*.exe or *.txt etc.).

In the following example, the browse dialog box displays all files with extensions .dll or .exe.

```
filename = create_browse_file_dialog( "*.dll;*.exe"  );
```



Browse

Look in: arch

File name: *.dll; *.exe

Files of type: All Files (*.*)

When the **Open** button is clicked, the name and path of the selected file is passed to the variable *filename*. If the **Cancel** button is clicked, an empty string (empty quotation marks) is passed to the variable *filename*.

Creating a Password Dialog Box

A password dialog box has two edit fields, an OK button, and a Cancel button. You supply the labels for the edit fields. The text that the user types into the edit fields during the interactive test run is saved to variables for analysis.

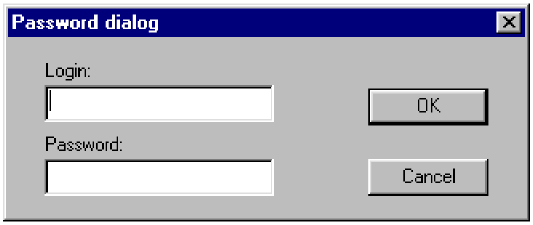
You use the TSL function **create_password_dialog** to create a password dialog box. This function has the following syntax:

create_password_dialog (*login*, *password*, *login_out*, *password_out* **);**

- *login* is the label of the first edit field, used for user-name input. If you specify an empty string (empty quotation marks), the default label "Login" is displayed.
- *password* is the label of the second edit field, used for password input. If you specify an empty string (empty quotation marks), the default label "Password" is displayed. When the user enters input into this edit field, the characters do not appear on the screen, but are represented by asterisks.
- *login_out* is the name of the parameter to which the contents of the first edit field (login) are passed. Use this parameter to verify the contents of the login edit field.
- *password_out* is the name of the parameter to which the contents of the second edit field (password) are passed. Use this parameter to verify the contents of the password edit field.

The following example shows a password dialog box created using the default edit field labels.

```
status = create_password_dialog ("", "", user_name, password);
```



If the **OK** button is clicked, the value 1 is passed to the variable *status*. If the **Cancel** button is clicked, the value 0 is passed to the variable *status* and the *login_out* and *password_out* parameters are assigned empty strings.

Part V

Running Tests

33

Understanding Test Runs

Once you have developed a test script, you run the test to check the behavior of your application.

This chapter describes:

About Understanding Test Runs

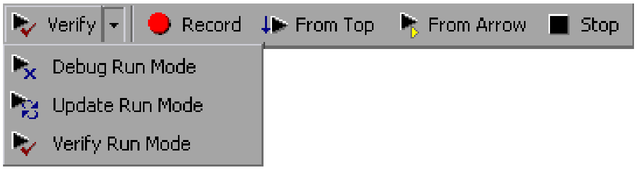
When you run a test, WinRunner interprets your test script, line by line. The execution arrow in the left margin of the test script marks each TSL statement as it is interpreted. As the test runs, WinRunner operates your application as though a person were at the controls.

You can run your tests in three modes:

- Verify run mode, to check your application
- Debug run mode, to debug your test script
- Update run mode, to update the expected results

Note: If you are working with WinRunner Runtime, you cannot run tests in Update mode.

You choose a run mode from the list on the Test toolbar. The Verify mode is the default run mode.



Use WinRunner's **Test** and **Debug** 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 5, "Working in the Global GUI Map File Mode."

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 35, "Running Batch Tests."

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. You can set the default run mode in the General Options dialog box.

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 34, "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 628.

Note: Before your run a test in Verify mode, you must have expected results for the checkpoints you created. If you need to update the expected results of your test, you must run the test in Update mode, as described on page 622.

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**.

Use WinRunner's debugging facilities when you debug a test script:

- ➤ Use the Step commands to control how your tests run. For more information, see Chapter 37, "Controlling Your Test Run."
- ➤ Set breakpoints at specified points in the test script to pause tests while they run. For more information, see Chapter 38, "Using Breakpoints."
- ➤ Use the Watch List to monitor variables in a test script while the test runs. For more information, see Chapter 39, "Monitoring Variables."
- ➤ Use the Call Chain to follow and navigate the test flow. For more information, see Chapter 28, "Calling Tests."

For more information about running a test script in Debug mode, see "Running a Test to Debug Your Test Script" on page 629.

Tip: You should change the timeout variables to zero while you debug your test scripts, to make them run more quickly. For more information on how to change these variables, see Chapter 41, "Setting Global Testing Options," and Chapter 44, "Setting Testing Options from a Test Script."

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 98 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 631.

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

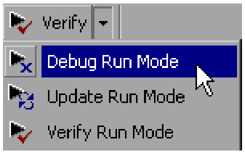
For more information about running a test script in Update mode, see "Running a Test to Update Expected Results" on page 630.

Setting the Run Mode for a Test

You use the Run mode toolbar button to set the run mode for a test.

To set the run mode for an open test:

1 Click the arrow next to the **Verify** toolbar button in the Test toolbar.



2 Select the run mode you want to use for the test. The icon and text in the toolbar button changes according to the run mode you select.

Setting the Default Run Mode

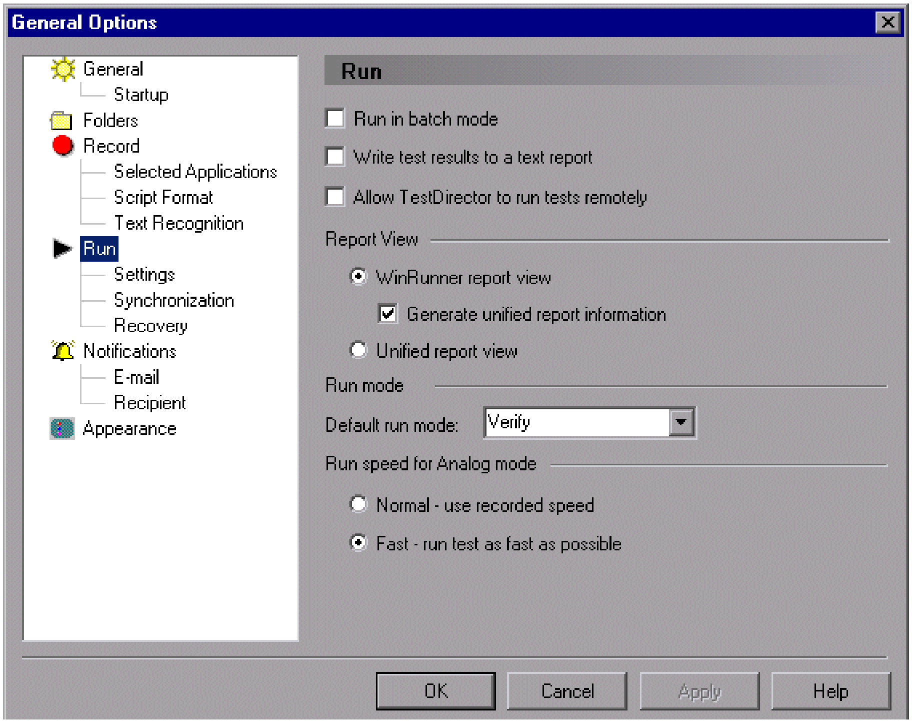
You can set the default run mode in the **Run** category of the General Options dialog box. The mode set here determines the mode in which each test opens.

For example, if you set **Debug** as the default run mode, then each test you open, opens in the Debug run mode. If you change the run mode for a particular test, that change remains in effect only while the test is open. If you save and close the test and then reopen it, the test again opens in the default run mode (**Debug**, in this example).

To set the default run mode:

1 Choose **Tools** > **General Options**. The General Options dialog box opens.

2 Select the **Run** category.



Run

□ Run in batch mode

□ Write test results to a text report

□ Allow TestDirector to run tests remotely

Report View

● WinRunner report view

☑ Generate unified report information

○ Unified report view

Run mode

Default run mode: Verify

Run speed for Analog mode

○ Normal - use recorded speed

● Fast - run test as fast as possible

3 Select a mode in the **Default run mode** box.

4 Click **OK** to save your changes and close the General Options dialog box.

Note: This option only applies to tests you open after you change the setting. It does not affect tests already open in WinRunner.

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 **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 **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 37, “Controlling Your Test Run.”

The following Step buttons are available:

Step button

Step Into button

The following Step commands are available:

- **Step** command
- **Step Into** command
- **Step Out** command
- **Step to Cursor** command

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 the Test Run with Testing Options" on page 639 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 30, "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 the Test Run with Testing Options" on page 639 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 43, "Customizing the WinRunner 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 OUT	CTRL LEFT + 7	Used in conjunction with **Step Into**—completes the execution of a called test or user-defined function.
STEP TO CURSOR	CTRL LEFT + F9	Runs a test from the line executed until the line marked by the insertion point.
PAUSE TEST RUN	PAUSE	Stops the test run after all previously interpreted TSL statements have been executed. Execution can be resumed from this point.
STOP TEST RUN	CTRL LEFT + F3	Stops the test run.

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.

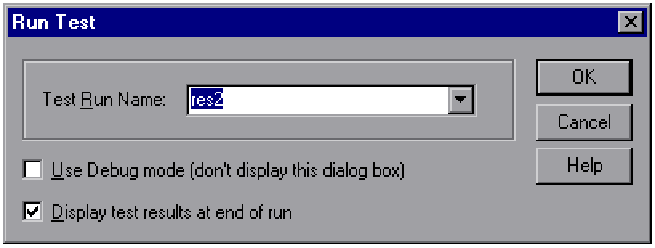
To run a test to check your application:

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 list of run modes on the Test toolbar.

3. Choose the appropriate **Test** 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: res2

□ Use Debug mode (don't display this dialog box)

☑ Display test results at end of run

4. You can save the test results under the default test run name. To use a different name, type in a new name or select an existing name from the list.

   To instruct WinRunner to display the test results automatically following the test run (the default), select the **Display test results at end of run** check box.

   Click **OK**. The Run Test dialog box closes and WinRunner runs the test according to the Run command you chose.

5. Test results are saved with the test run name you specified.

Running a Test to Debug Your Test Script

When you run a test to debug your test script, WinRunner compares the current results with the expected results. Any differences are saved in the debug results 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 Test toolbar.

3 Choose the appropriate **Run** or **Debug** menu command.

To execute the entire test, choose **Test** > **Run from Top** or click the **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:

Test > **Run from Arrow**

Test >**Run Minimized** > **From Arrow**

Debug > **Step**

Debug >**Step Into**

Debug >**Step Out**

Debug >**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 Test toolbar.

3 Choose the appropriate **Test** menu command.

To update the entire set of expected results, choose **Test** > **Run from Top** or click the **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:

Test > **Run from Arrow**

Test > **Run Minimized** > **From Arrow**

Debug >**Step**

Debug >**Step Into**

Debug >**Step Out**

Debug >**Step to Cursor**

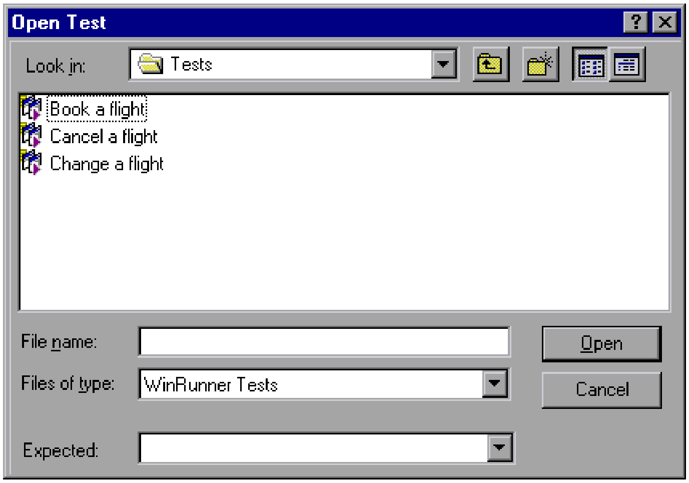
WinRunner runs the test according to the **Test** menu command you chose and updates the expected results. The default folder for expected results is *exp*.

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.



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 Test toolbar.

5. Choose **Test** > **Run from Top** or click the **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.

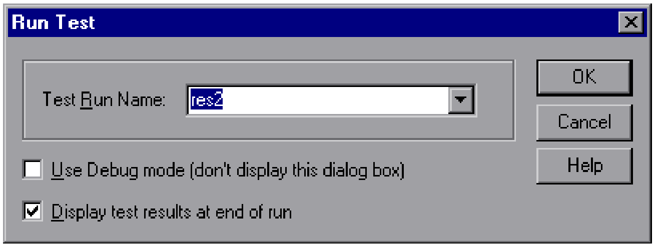
---

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 Test toolbar.

5 Choose the appropriate **Test** menu command. The Run Test dialog box opens, displaying a default test run name for the verification results—for example, *res2*.



Run Test

Test Run Name: res2

□ Use Debug mode (don't display this dialog box)

☑ Display test results at end of run

6 Click **OK** to begin the test run, 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 **Test** menu command you chose and saves the test results in the folder you specified.

Running a Test to Check Date Operations

Once you have created a test that checks date operations, as described in Chapter 20, “Checking Dates,” you run your test to check how your application responds to date information in your test.

Note that the **Enable date operations** option must be selected in the **General** category of the Options dialog box when you run a test with date checkpoints. Otherwise, the date checkpoints will fail.

When you run a test that checks date operations, WinRunner interprets the test script line-by-line and performs the required operations on your application. At each checkpoint in the test script, it compares the expected dates with the actual dates in your application.

Before you run your test, you first specify date operations settings and the general run mode of the script.

Date operations run mode settings specify:

- *Date format*, to determine whether to use the script's original date formats or to convert dates to new formats.
- *Aging*, to determine whether or not to age the dates in the script.

  You can age dates incrementally (by specifying the years, months, and days by which you want to age the dates) or statically (by defining a specific date).

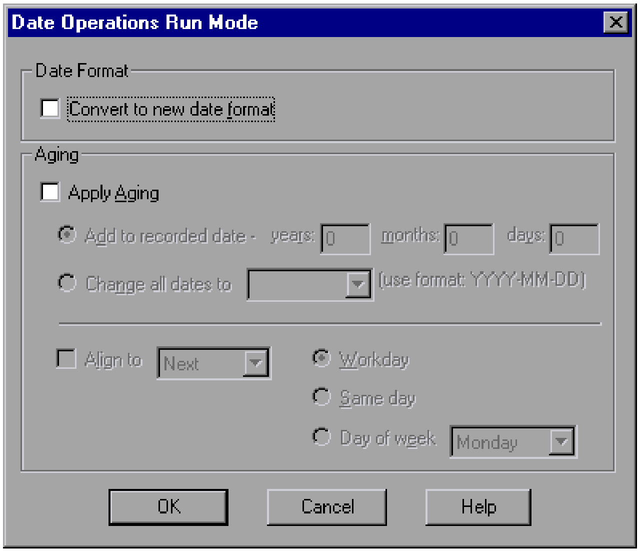
  The general run mode settings, Verify, Debug, and Update, are described earlier in this chapter. Note that during a test run in Update mode, dates in the script are not aged or translated to a new format.

Setting the Date Operations Run Mode

Before you run a test that checks date operations, you set the date operations run mode.

To set the date operations run mode:

1 Choose **Tools** > **Date** > **Run Mode** (available only when the **Enable date operations** check box is selected in the **General** category of the General Options dialog box). The Date Operations Run Mode dialog box opens.



You can also open this dialog box from the Run Test dialog box by clicking the **Change** button (only when the **Enable date operations** check box is selected in the **General** category of the General Options dialog box). For more information on the General Options dialog box, see Chapter 41, "Setting Global Testing Options." For more information on the Run Test dialog box, see "Running Tests to Check Date Operations" on page 635.

2 If you are running the test on an application that was converted to a new date format, select the **Convert to new date format** check box.

3 If you want to run the test with aging, select the **Apply Aging** check box and do one of the following:

➤ To increment all dates, click **Add to recorded date** and specify the years, months or days. You can also align dates to a particular day by clicking the **Align to** check box and specifying the day.

➤ To change all dates to a specific date, click **Change all dates to** and select a date from the list.

4 Click **OK**.

Note: When you run a test, you can override the options you specified in the Date Operations Run Mode dialog box. For more information, see "Overriding Date Settings" on page 412.

Running Tests to Check Date Operations

After you set the date operations run mode, you can run your test script.

To run a test that checks date operations:

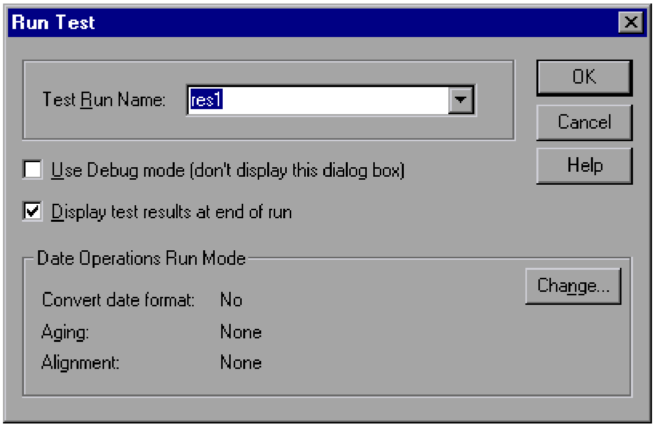
1 If the test is not already open, open it.

2 Choose a general run mode (Verify, Debug, or Update) from the list of modes on the Test toolbar.

3 Choose the appropriate **Test** menu command or click one of the **Run** buttons. For more information on Run commands, see "WinRunner Run Commands" on page 625.

Note that in *Update* mode, dates in the script are not aged or translated to a new format. In *Debug* mode the test script immediately starts to run using the date operations run mode settings defined in the Date Operations Run Mode dialog box.

If you selected *Verify* mode, the Run Test dialog box for date operations opens.



Test Run Name: res1

□ Use Debug mode (don't display this dialog box)

☑ Display test results at end of run

Date Operations Run Mode

Convert date format: No

Aging: None

Alignment: None

4 Assign a name to the test run. Use the default name appearing in the **Test Run Name** field, or type in a new name.

5 If you want to change the date operations run mode settings, click **Change** and specify the date operations run mode settings.

6 Click **OK** to close the dialog box and run the test.

Changing Date Operations Run Mode Settings with TSL

You can set conditions for running a test checking date operations using the following TSL functions:

➤ The **date_align_day** function ages dates to a specified day of the week or type of day. It has the following syntax:

date_align_day (*align_mode, day_in_week* **);**

- The **date_disable_format** function disables a date format. It has the following syntax:

  date_disable_format (*format* **);**

- The **date_enable_format** function enables a date format. It has the following syntax:

  date_enable_format (*format* **);**

- The **date_leading_zero** function determines whether to add a zero before single-digit numbers when aging and translating dates. It has the following syntax:

  date_leading_zero (*mode* **);**

- The **date_set_aging** function ages the test script. It has the following syntax:

  date_set_aging (*format, type, days, months, years* **);**

- The **date_set_run_mode** function sets the date operations run mode. It has the following syntax:

  date_set_run_mode (*mode* **);**

- The **date_set_year_limits** function sets the minimum and maximum years valid for date verification and aging. It has the following syntax:

  date_set_year_limits (*min_year, max_year* **);**

- The **date_set_year_threshold** function sets the year threshold (cut-year point). If the threshold is 60, all years from 60 to 99 are recognized as 20th century dates and all dates from 0 to 59 are recognized as 21st century dates. This function has the following syntax:

  date_set_year_threshold (*number* **);**

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

Controlling the Test Run 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 **Tools** > **General Options** to open this dialog box. You can also set testing options from within a test script using the **setvar** function.

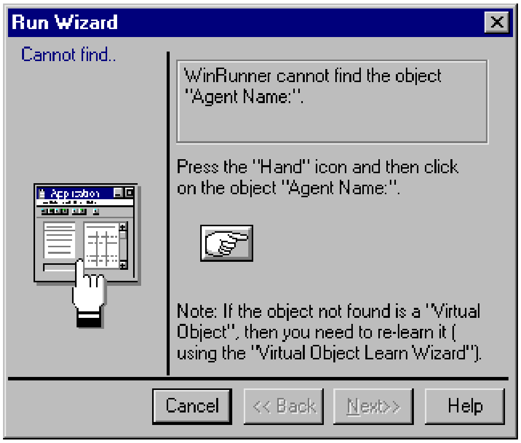
Each testing option has a default value. For example, the default for the threshold for difference between bitmaps option (that defines the minimum number of pixels that constitute a bitmap mismatch) is 0. It can be set globally in the **Run** > **Settings** category of the General Options dialog box. For a more comprehensive discussion of setting testing options globally, see Chapter 41, "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 44, "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.

Solving Common Test Run Problems

When you run your Context Sensitive test, WinRunner may open the Run wizard. Generally, the Run wizard opens when WinRunner has trouble locating an object or a window in your application. It displays a message similar to the one below.



WinRunner cannot find the object "Agent Name:".

Press the "Hand" icon and then click on the object "Agent Name:".

Note: If the object not found is a "Virtual Object", then you need to re-learn it (using the "Virtual Object Learn Wizard").

There are several possible causes and solutions:

Possible Causes	**Possible Solutions**
You were working with the temporary GUI map, which you did not save when you exited WinRunner: When you record in an application, WinRunner learns the GUI objects on which you record. Unless you specify otherwise, this information is stored in the temporary GUI map file, which is cleared whenever you exit WinRunner.	WinRunner should relearn your application, so that the logical names and physical descriptions of the GUI objects are stored in the GUI map. When you are done, make sure to save the GUI map file. When you start your test, make sure to *load* your GUI map file. These steps are described in Chapter 5, "Working in the Global GUI Map File Mode."
You saved the GUI map file, but it is not loaded.	Load the GUI file for your test. You can load the file manually each time with the GUI Map Editor, or you can add a **GUI_load** statement to the beginning of your test script. For more information, see Chapter 5, "Working in the Global GUI Map File Mode."
The object is not identified during a test run because it has a dynamic label. For example, you may be testing an application that contains an object with a varying label, such as any window that contains the application name followed by the active document name in the title. (In the sample Flight Reservation application, the "Fax Order" window also has a varying label.)	Use a regular expression to enable WinRunner to recognize objects based on a portion of its physical description. For more information on regular expressions, see Chapter 25, "Using Regular Expressions."
	Use the GUI Map Configuration dialog box to change the physical properties that WinRunner uses to recognize the problematic object. For more information on GUI Map configuration, Chapter 9, "Configuring the GUI Map."
The physical description of the object/window does not match the physical description in the GUI map.	Modify the physical description in the GUI map, as described in "Modifying Logical Names and Physical Descriptions" on page 81.

Possible Causes	Possible Solutions
The logical name of the object/window in the test script does not match the logical name in the GUI map.	Modify the logical name of the object/window in the GUI map, as described in "Modifying Logical Names and Physical Descriptions" on page 81.
	Modify the logical name of the object/window manually in the test script.
The object/window has a different number of obligatory or optional properties (in the GUI map configuration) than in the GUI map.	In the Configure Class dialog box, configure the obligatory or optional properties which are learned by WinRunner for that class of object, so they will match the physical description in the GUI map, as described in "Configuring a Standard or Custom Class" on page 115.
	WinRunner should relearn the object/window in the GUI map so that it will learn the obligatory and optional properties configured for that class of object, as described in Chapter 5, "Working in the Global GUI Map File Mode."

Tip: WinRunner can learn your application systematically from the GUI Map Editor before you start recording on objects within your application. For more information, see Chapter 5, "Working in the Global GUI Map File Mode."

Note: For additional information on solving GUI map problems while running a test, see "Guidelines for Working in the Global GUI Map File Mode," on page 66.

34

Analyzing Test Results

After you run a test, you can view a report of all the major events that occurred during the test run in the Test Results Window. You can view your results in the standard WinRunner report view or in the Unified report view.

This chapter describes:

About Analyzing Test Results

After you run a test, you can view the results in the Test Results window. The appearance of this window depends on the Report View option you select in the **Run** category of the General Options dialog box.

There are two types of Test Results Views:

- **WinRunner report view**—Displays the test results in a Windows-style viewer.

  If you run a test that includes a call to a QuickTest test, the WinRunner report view displays only basic information about the results of the QuickTest test.

  When running tests that call QuickTest tests, it is recommended to use the Unified report view.

- **Unified report view**—Displays the results in an HTML-style viewer.

  The unified TestFusion report viewer is identical to the style used for QuickTest Professional test results.

  If you run a test that includes a call to a QuickTest test (version 6.5 or later), the unified report view enables you to view detailed results of each step in the called QuickTest test.

Regardless of the selected report view, the test results window always 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.

Understanding the Unified Report View Results Window

If you are new to WinRunner, or you are integrating WinRunner and QuickTest tests, it is recommended to use the Unified Report view. For information on analyzing the results of a called QuickTest test, see Chapter 47, "Integrating with QuickTest Professional."

To view the unified report, choose **Tools** > **General Options**. In the **Run** category, confirm that **Unified report view** is selected.

Note: You can display the unified report view for a test only if either the **Unified report view** or the **Generate unified report information** option was selected when you ran the test. If you ran a test with **WinRunner report view** selected and **Generate unified report information** cleared, then you cannot view the unified report for that test run.

To open the Test Results window, choose **Tools** > **Test Results** or click the **Test Results** button. The WinRunner Test Results window opens in the unified report view.

Test name and results location

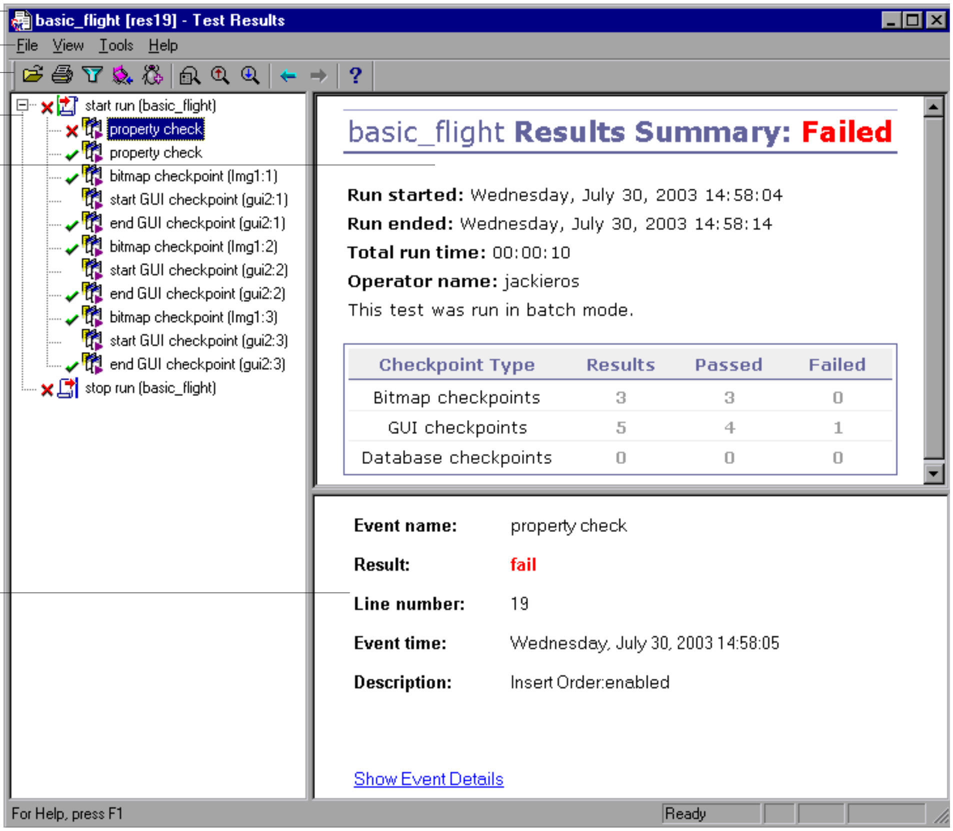
Menu bar

Toolbar

Results tree

Test summary

Event summary



start run (basic_flight)
- ✗ property check
- ✓ property check
- ✓ bitmap checkpoint (Img1:1)
- start GUI checkpoint (gui2:1)
- ✓ end GUI checkpoint (gui2:1)
- ✓ bitmap checkpoint (Img1:2)
- start GUI checkpoint (gui2:2)
- ✓ end GUI checkpoint (gui2:2)
- ✓ bitmap checkpoint (Img1:3)
- start GUI checkpoint (gui2:3)
- ✓ end GUI checkpoint (gui2:3)

✗ stop run (basic_flight)

basic_flight **Results Summary: Failed**

Run started: Wednesday, July 30, 2003 14:58:04
Run ended: Wednesday, July 30, 2003 14:58:14
Total run time: 00:00:10
Operator name: jackieros
This test was run in batch mode.

Checkpoint Type	Results	Passed	Failed
Bitmap checkpoints	3	3	0
GUI checkpoints	5	4	1
Database checkpoints	0	0	0

Event name: property check
Result: fail
Line number: 19
Event time: Wednesday, July 30, 2003 14:58:05
Description: Insert Order:enabled

Show Event Details

For more information on opening the test results window, see "Viewing the Results of a Test Run" on page 658.

Test Name and Results Location

The Unified Report View titlebar displays the name of the test and the test results folder.

Menu Bar and Toolbar

The menu bar contains the options that you can use to analyze the test results. Several of these options can also be performed using the corresponding Test Results toolbar button, as indicated below.

- **File menu**—Contains options for opening and printing test results, and exiting the Test Results window.
  - **Open**—Opens the Open Test Results dialog box, which enables you to select a test and open the most recent results for that test.
  - **Print**—Opens the Print dialog box, which enables you to print the selected display or the entire test results.
  - **Print Setup**—Opens the Print Setup dialog box, which enables you to select printer options.
  - **Recent Files**—Displays the four most recent files that were opened in the Test Results window.
  - **Exit**—Closes the Test Results window.
- **View**—Contains options for viewing test results window components and analyzing specific elements of the test results
  - **Test Results Toolbar**—Displays or hides the test results toolbar.
  - **Status Bar**—Displays or hides the test results status bar.
  - **Filters**—Opens the Filters dialog box, which enables you to choose which types of test steps you want to view.
  - **Expand All**—Expands all step nodes in the test tree.
  - **Collapse All**—Collapses all step nodes in the test tree.

- **Tools**—Contains options for connecting to and adding defects to TestDirector and for navigating the test to find steps with a specified result status.
  - **Add Defect**—If the Test Results window is already connected to TestDirector, selecting this option opens the TestDirector Add Defect dialog box, which enables you to add a defect to the TestDirector project.

    If you are not yet connected, choosing this option opens the TestDirector Connection dialog box. After you connect to TestDirector, the TestDirector Add Defect dialog box opens.
  - **TestDirector Connection**—Opens the TestDirector Connection dialog box, which enables you to connect the Test Results window to a TestDirector project.

Note: The unified report viewer is a standalone application. Therefore, even if WinRunner is connected to TestDirector, you must still connect the Test Results window to the TestDirector project in order to report bugs from the Test Results window.

  - **Find**—Opens the Find dialog box, which enables you to navigate up or down through your test to find result steps with the selected status.
- **Help**—Contains options for accessing additional information about the Test Results window.
  - **Contents and Index**—Opens the Test Results Help file.
  - **Support Information**—Opens the Mercury Interactive Customer Support Help file.
  - **About Test Results**—Opens a window with summary information about the Test Results application.

Results Tree

The Results tree shows a hierarchical view of all events performed during the test run. Selecting an event in the results tree displays additional details of the event in the Event Summary pane. You can expand and collapse the tree or individual nodes in the tree. You can also use the **Filter** and **Find** options for easier navigation.

Test Summary

Contains overview information about the test run including the run start time, run end time, total test run time, user name, and a summary of checkpoint results.

Note: Unlike the WinRunner report view, the Unified report view counts single-property checks in the GUI checkpoint summary. Therefore, the total number of GUI checkpoints in the Unified report view may differ from the number displayed in the WinRunner report view.

Event Summary

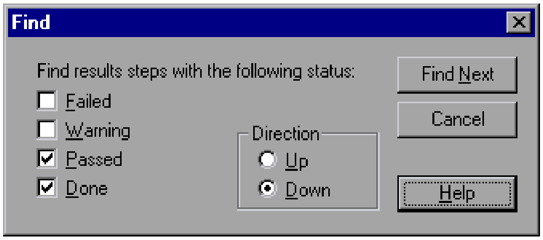
Contains summary information about the event that is currently selected in the results tree, including the event type, status, line number, event time, and a basic description of what was checked.

For checkpoints (including single-property checks), the Event Summary also includes a link to the event details. For example, if you click the **Show Event Details** link for a bitmap checkpoint, then the expected, actual, and difference images open. If you click the link for a GUI Checkpoint, the GUI Checkpoint Results window opens.

Note: To view checkpoint details, WinRunner must be installed on the Test Results computer.

Finding Results Steps

The Find dialog box enables you to find specified steps such as errors or warnings from within the Test Results. You can select a combination of statuses to find, for example, steps that are **Passed** and **Done**.

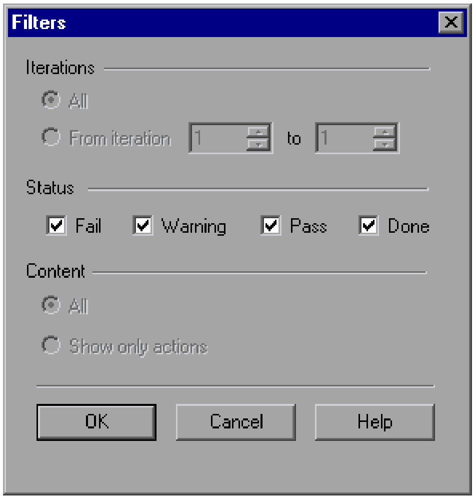


The following options are available:

Option	Description
Failed	Finds a failed step in the test results.
Warning	Find a step where a warning was issued.
Passed	Finds a passed step in the test results.
Done	Finds a step that has finished its run.
Direction	Indicates whether to search **Up** or **Down** within the steps of the test results.

Filtering Test Results

The Filters dialog box enables you to filter which results are displayed in the test results tree, according to their status.



Filters

Iterations

● All

○ From iteration 1 to 1

Status

☑ Fail ☑ Warning ☑ Pass ☑ Done

Content

● All

○ Show only actions

Note: The **Iterations** and **Content** options are only available from QuickTest.

The following options are available:

Option	Description
Status	• **Fail**—Displays the test results for the steps that failed. • **Warning**—Displays the test results for the steps with a Warning status (steps that did not pass, but did not cause the test to fail). • **Pass**—Displays the test results for the steps that passed. • **Done**—Displays the test results for the steps with a Done status (steps that were performed successfully but did not receive a pass, fail, or warning status).

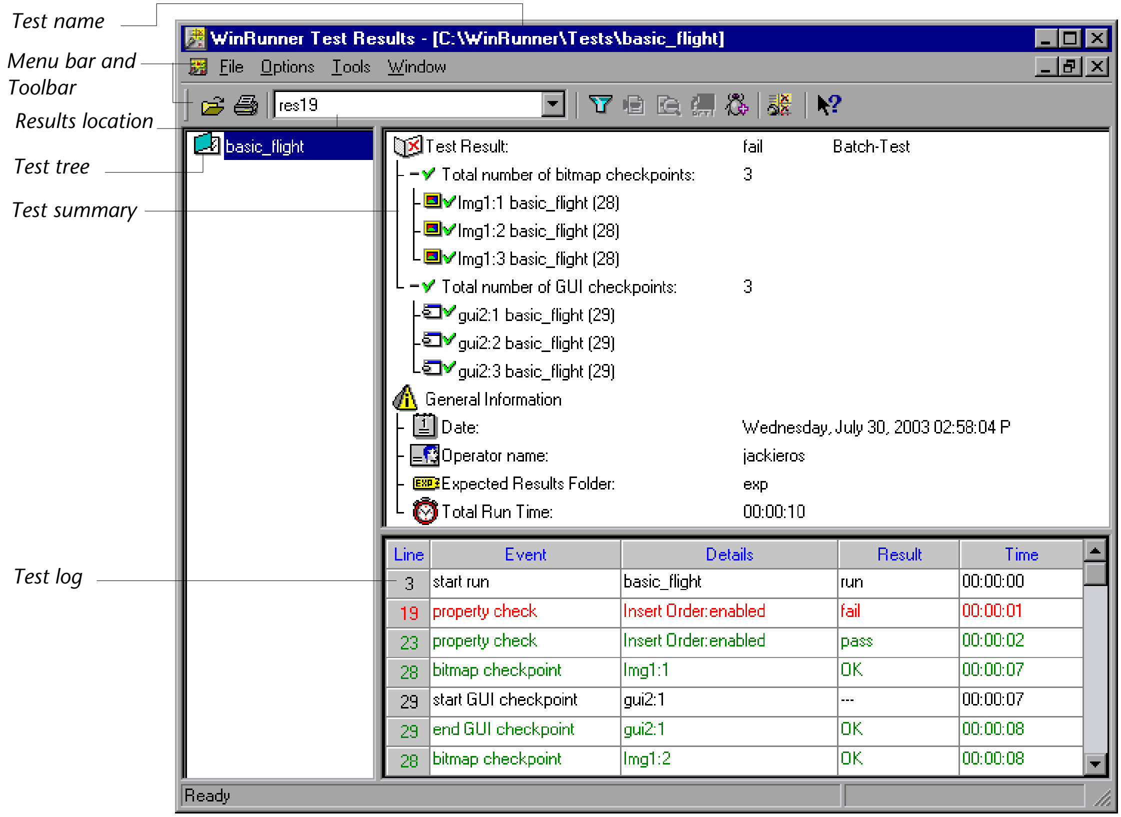
Understanding the WinRunner Report View Results Window

If you have worked with previous versions of WinRunner, and you are not analyzing the results of a called QuickTest test, you may feel more comfortable using the **WinRunner report view**.

To view the WinRunner report, choose **Tools** > **General Options**. In the **Run** category, confirm that **WinRunner report view** is selected.

Note: By default, the WinRunner report is displayed and unified report files are created so that you can choose to view the Unified report for the test run at a later time. If you do not want WinRunner to generate unified report files, clear the Generate unified report information option.

To open the Test Results window, choose **Tools** > **Test Results** or click the **Test Results** button. The WinRunner Test Results window opens in the WinRunner report view.



WinRunner Test Results - [C:\WinRunner\Tests\basic_flight]

```
Test Result:                                  fail        Batch-Test
 - ✓ Total number of bitmap checkpoints:      3
     ✓Img1:1 basic_flight (28)
     ✓Img1:2 basic_flight (28)
     ✓Img1:3 basic_flight (28)
 - ✓ Total number of GUI checkpoints:         3
     ✓gui2:1 basic_flight (29)
     ✓gui2:2 basic_flight (29)
     ✓gui2:3 basic_flight (29)
General Information
   Date:                                      Wednesday, July 30, 2003 02:58:04 P
   Operator name:                             jackieros
   Expected Results Folder:                   exp
   Total Run Time:                            00:00:10
```

Line	Event	Details	Result	Time
3	start run	basic_flight	run	00:00:00
19	property check	Insert Order:enabled	fail	00:00:01
23	property check	Insert Order:enabled	pass	00:00:02
28	bitmap checkpoint	Img1:1	OK	00:00:07
29	start GUI checkpoint	gui2:1	---	00:00:07
29	end GUI checkpoint	gui2:1	OK	00:00:08
28	bitmap checkpoint	Img1:2	OK	00:00:08

For more information on opening the Test Results window, see "Viewing the Results of a Test Run" on page 658.

Test Name

The Test Results title bar displays the full path of the test.

Menu Bar and Toolbar

The menu bar contains the options that you can use to analyze the test results. Several of these options can also be performed using the corresponding **Test Results** toolbar button, as indicated below.

- **File menu**—Contains options for opening, closing, and printing test results, and exiting the Test Results window.
  - **Open**—Enables you to select a test and open the most recent results for that test.
  - **Close**—Closes the active test results window.
  - **Print**—Opens the Print dialog box, enabling you to print a text-only version of the information displayed in the test summary and test log panes.
  - **Exit**—Exits the WinRunner Test Results viewer.
- **Options menu**—Contains options for viewing and analyzing specific elements of the test results.
  - **Filters**—Opens the Filters dialog box, which enables you to select which events are included in the test log.
  - **Bitmap Controls**—Opens the Bitmap Controls dialog box, which enables you to select which images to include in the bitmaps display for bitmap checkpoints. For more information, see “Analyzing the Results of a Bitmap Checkpoint” on page 677.
  - **Show TSL**—Opens the WinRunner test in the WinRunner window (if it is not already open) and highlights the line in the WinRunner test corresponding to the results line currently selected in the test log.
  - **Display**—Opens the results details for the currently selected line in the test log. Choosing this option is equivalent to double-clicking the line in the test log.
  - **Update**—Updates the expected data for the selected bitmap, GUI, or database checkpoint to match the actual results of the selected checkpoint. Enabled only when a failed bitmap, GUI, or database checkpoint is selected.

  - **Mismatches Only**—Hides bitmap, database, and GUI checkpoint events with **Pass** or **OK** status. This option does not affect property checks or other non-checkpoint events.

- **Tools menu**—Contains options for generating text-only results files and reporting defects to TestDirector.
  - **Text Report**—Generates and displays a text-only version of the test results for the active test results window.
  - **Report Bug**—Reports a bug for the selected event in the test log to the TestDirector project to which you are currently connected. (This option is enabled only when you are connected to a TestDirector project).
- **Window menu**—Contains options for opening additional test results windows and arranging them within the main Test Results window.
  - **New Window**—Opens a new Test Results window containing a copy of the results of the currently active results window. To view the results for a different run of the displayed results, select the results name from the **Results location** box.
  - **Cascade**—Displays all open Test Results windows in a cascading display.
  - **Tile**—Horizontally tiles all open Test Results windows.
  - **Arrange Icons**—Arranges minimized test results icons in the Test Results window.

Results Location

The **results location** 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.

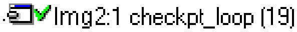
Test Summary

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 branch in the tree 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. Each listing contains important information about the checkpoint. For example:



✓ Img2:1 checkpt_loop (19)

provides the following information:

Element	Description
✓	Indicates that the checkpoint passed.
Img2	The name of the captured bitmap file.
:1	The first time this checkpoint was run in the script.
checkpt_loop	The name of the test.
(19)	The 19th line in the test script contains the **obj_check_bitmap** or **win_check_bitmap** statement.

Double-click the bitmap checkpoint listing to display the contents of the bitmap checkpoint. For more information, see "Analyzing the Results of a Bitmap Checkpoint" on page 677.

Total number of GUI checkpoints: The total number of GUI and database checkpoints that occurred during the test run.

Note: Unlike the Unified report view, the WinRunner report view does not count single-property checks in the GUI checkpoint summary. Therefore, the total number of GUI checkpoints in the WinRunner report view may differ from the number displayed in the Unified report view.

Double-click to view a detailed list of the checkpoints. For example, the elements in the listing

```
gui1:4 checkpt_loop (12)
```

have the following meanings:

Element	Description
gui1	The name of the expected results file.
:4	The fourth time this checkpoint was run in the script.
checkpt_loop	The name of the test.
(12)	The12th line in the test script 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 “Analyzing the Results of a GUI Checkpoint” on page 667.

➤ 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.

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, calls to other tests, and run time errors.

- A row describing a mismatch or failure appears in red; a row describing a successful event appears in green.
- The **Line** column displays the line number in the test script at which the event occurs.
- The **Event** column describes the event, such as the start or end of a checkpoint or of the entire test.
- The **Details** column provides specific information about the event, such as the name of the test (for starting or stopping a test), the name of the expected results file (for a checkpoint), or a message (for a **tl_step** statement).
- The **Result** column displays whether the event passed or failed, if applicable.
- The **Time** column displays the amount of time elapsed (in hours:minutes:seconds) from when the test started running until the start of the event.

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.

- For a call to another test in batch mode, you can view whether the **call** statement passed. Note that even though a **call** statement is successful, the called test itself may fail, based on the usual criteria for tests failing. You can set criteria for failing a test in the **Run** > **Settings** category of the General Options dialog box. For additional information, see Chapter 41, "Setting Global Testing Options."

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. Confirm that the report view you prefer is selected in the **Run** category of the General Options dialog box. For more information, see "About Analyzing Test Results" on page 644 and "Setting Test Run Options" on page 793.
2. 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, click the **Open** button or choose **File** > **Open**. In the Open Test Results dialog box, select or browse to the test whose results you want to view.

Note: If you are browsing to a test from the Unified report view, confirm that **WinRunner Tests** is selected as the test type in the **Files of type** edit box.

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 33, "Understanding Test Runs."

3 By default, the Test Results window displays the results of the most recently executed test run.

To view other test run results:

- ➤ In the Unified report view—Click the **Open** button or choose **File** > **Open** and select a test run from the Open Test Results dialog box. For more information, see "Opening Test Results to View a Selected Test Run" on page 661.
- ➤ In the WinRunner report view—Click the **Results location** box and select a test run.

4 To view a text version of a report, display the WinRunner report view and choose **Tools** > **Text Report** from the Test Results window. The report is displayed in a Notepad window.

5 To view only specific types of results in the events column in the test log, choose **Options** > **Filters** or click the **Filters** button.

6 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.

7 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.

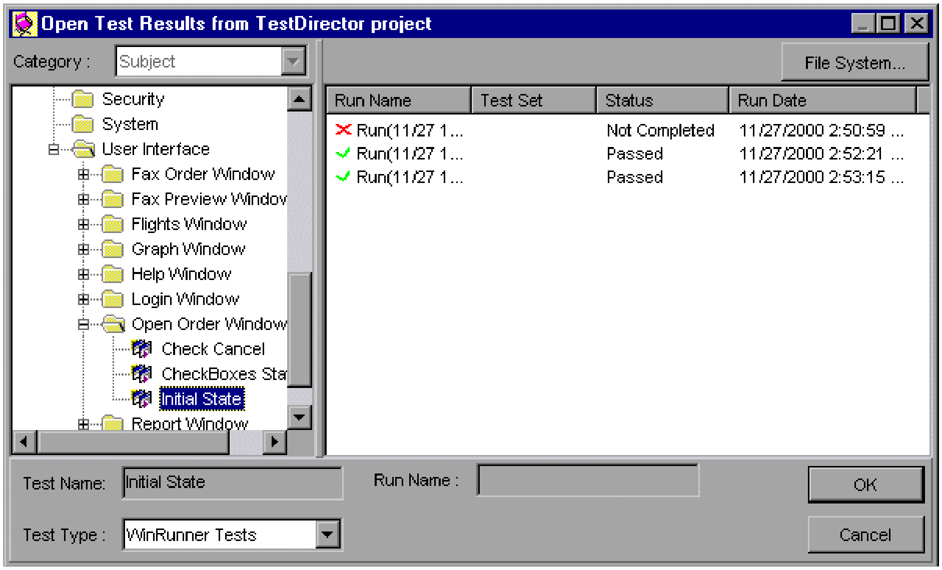
2 Connect to TestDirector:

- ➤ In the Unified report view—Click the **TestDirector Connection** button or choose **Tools** > **TestDirector Connection**.
- ➤ In the WinRunner report view—Switch to the WinRunner main window and choose **Tools** > **TestDirector Connection**.

3 Select the TestDirector test results:

➤ In the Unified report view—Choose **File** > **Open**. The Open Test Results dialog box displays results for the test currently open in the Test Results Window. If you want to view results for a different test, click **Browse**. The Open Test Results from TestDirector Project dialog box opens and displays the test plan tree.

➤ In the WinRunner report view—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

- Security
- System
- User Interface
  - Fax Order Window
  - Fax Preview Windov
  - Flights Window
  - Graph Window
  - Help Window
  - Login Window
  - Open Order Window
    - Check Cancel
    - CheckBoxes Sta
    - Initial State
  - Report Window

Run Name	Test Set	Status	Run Date
✗ Run(11/27 1...		Not Completed	11/27/2000 2:50:59 ...
✓ Run(11/27 1...		Passed	11/27/2000 2:52:21 ...
✓ Run(11/27 1...		Passed	11/27/2000 2:53:15 ...

Test Name: Initial State

Run Name :

Test Type : WinRunner Tests

4 In the **Test Type** box, select **WinRunner Tests**, **WinRunner Batch Tests**, or **All Tests**.

5 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.

6 Select a test run to view.

The **Run Name** column contains the names of the test runs and displays whether your test run passed or failed. (If you open this dialog box from the WinRunner report view, the Run Name of the selected run is also displayed in the read-only **Run Name** edit box.)

The **Test Set** column contains the names of the test sets.

Entries in the **Status** column indicate whether the test passed or failed.

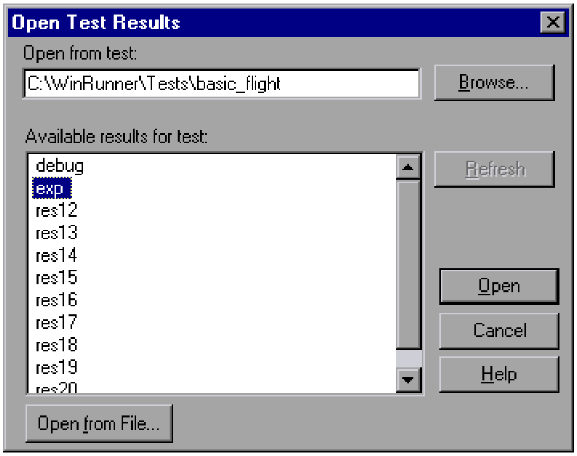
The **Run Date** column displays the date and time when the test set was run.

7 Click **OK** to view the results of the selected test.

Note: For more information on viewing the results of a test run from a TestDirector database, see Chapter 48, "Managing the Testing Process."

Opening Test Results to View a Selected Test Run

You can view the saved results for the current test, or you can view the saved results for other tests. You select the test results to open for viewing from the Open Test Results dialog box.



Open Test Results

Open from test: C:\WinRunner\Tests\basic_flight

Available results for test:

debug
exp
res12
res13
res14
res15
res16
res17
res18
res19
res20

The results of test runs for the currently open test are listed. To view one of the results sets, select it and click **Open**.

Tip: To update the results list after you change the specified test path, click **Refresh**.

To view results of test runs for other tests, you can search by test within WinRunner or by unified result (.qtp) files in your file system.

To search for results by test:

1. In the Open Test Results dialog box, enter the path of the test folder, or click **Browse** to open the Open Test dialog box.
2. In the **Files of type** box, select **WinRunner Tests**.
3. Find and highlight the test whose results you want to view, and click **Open**.
4. In the Open Test Results dialog box, highlight the test result set you want to view, and click **Open**.

To search for results by test result files:

1. From the Open Test Results dialog box, click the **Open from File** button to open the Select Results File dialog box.
2. Browse to the folder where the test results are stored. By default, the results folder is named <**TestName**>**\res*X*\Report**, where *X* is the number ID of the test results.
3. Highlight the unified test results report (.qtp) file you want to view, and click **Open**.

Connecting to TestDirector from the Test Results Window

To manually submit bugs to TestDirector from the Test Results window or to view test results stored in TestDirector, you must be connected to TestDirector.

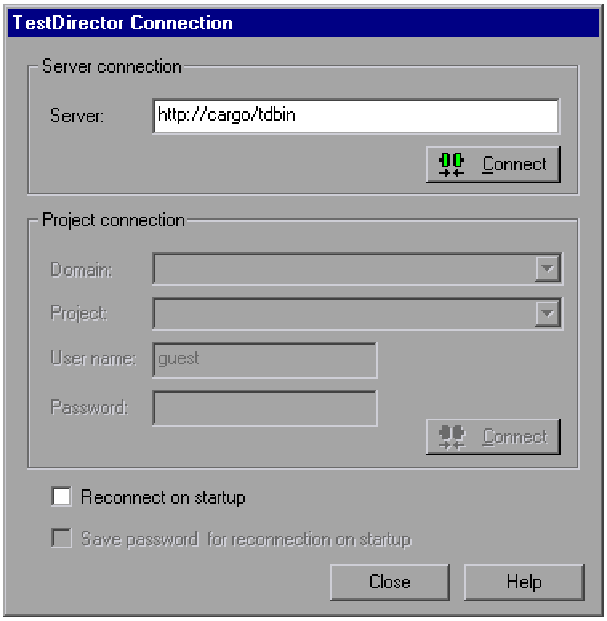
The connection process has two stages. First, you connect the WinRunner unified report to a local or remote TestDirector Web server. This server handles the connections between WinRunner and the TestDirector project.

Next, you choose the project in which you want to report the defects.

Note that TestDirector projects are password protected, so you must provide a user name and a password.

To connect the WinRunner unified report to TestDirector:

1 Choose **Tools** > **TestDirector Connection**. The TestDirector Connection dialog box opens.



TestDirector Connection

Server connection

Server: http://cargo/tdbin

Connect

Project connection

Domain:

Project:

User name: guest

Password:

Connect

□ Reconnect on startup

□ Save password for reconnection on startup

Close | Help

2 In the **Server** box, type the URL address of the Web server where TestDirector is installed.

Note: You can choose a Web server accessible via a Local Area Network (LAN) or a Wide Area Network (WAN).

---

3 Click **Connect**.

Once the connection to the server is established, the server name is displayed in read-only format in the Server box.

4. If you are connecting to a project in TestDirector 7.5 or later, in the **Domain** box, select the domain which contains the TestDirector project.

   If you are connecting to a project in TestDirector 7.2, skip this step.

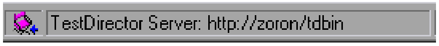
5. In the **Project** box, select the desired project with which you want to work.
6. In the **User name** box, type a user name for opening the selected project.
7. In the **Password** box, type the password.
8. Click **Connect** to connect the WinRunner unified report to the selected project.

   Once the connection to the selected project is established, the project name is displayed in read-only format in the Project box.

9. To automatically reconnect to the TestDirector server and the selected project the next time you open WinRunner or the WinRunner unified report, select the **Reconnect on startup** check box.
10. If you select the **Reconnect on startup** check box, 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.

11. Click **Close** to close the TestDirector Connection dialog box. The TestDirector icon and the address of the TestDirector server are displayed in the status bar to indicate that the WinRunner unified report is currently connected to a TestDirector project.



TestDirector Server: http://zoron/tdbin

Tip: You can open the TestDirector Connection dialog box by double-clicking the **TestDirector** icon in the status bar.

You can disconnect from a TestDirector project and/or server. Note that if you disconnect the WinRunner unified report from a TestDirector server without first disconnecting from a project, the WinRunner unified report's connection to that project database is automatically disconnected.

Viewing Checkpoint Results

You can view the results of a specific checkpoint in your test. A checkpoint helps you to identify specific changes in the behavior of objects in your application.

The procedure for displaying checkpoint results details varies depending on the report view you are using.

To display the results of a checkpoint from the Unified report view:

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 results tree, look for the checkpoint you want to check.
   - Failed checks are preceded by a red **X**; passed checks are preceded by a green check mark.
   - Each checkpoint node specifies the checkpoint type. All checkpoint nodes except single-property checks also list the name and iteration of the checkpoint, which helps you identify the node you want to view. For example:

     ```
     end GUI checkpoint (gui3:2)
     ```

     gui3 is the name of the expected results file for the checkpoint. The 2 after the colon indicates that this is the second time this checkpoint was run in the script (for example, the second iteration in a loop).
3. Click the node for the checkpoint you want to analyze. Basic details about the checkpoint are displayed in the **Event Summary** pane.
4. In the Event Summary pane, click the **Show Event Details** link. The relevant dialog box opens.
5. Click **OK** to close the dialog box.

The remaining sections in this chapter describe the results information that is provided for various event types.

To display the results of a checkpoint from the WinRunner report view:

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 the checkpoint you want to check.
   - Failed checks appear in red; passed checks appear in green.
   - The **Details** column displays information about the checkpoint that helps you identify each one. For example:

     ```
     gui3:2
     ```

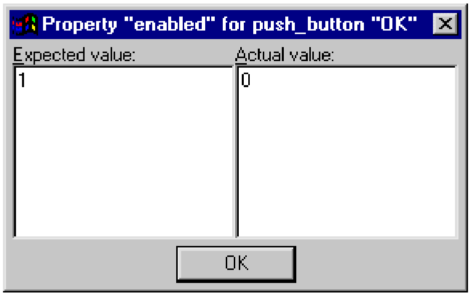
     gui3 is the name of the expected results file for the checkpoint. The 2 after the colon indicates that this is the second time this checkpoint was run in the script (for example, the second iteration in a loop).
3. Double-click the appropriate entry in the test log. Alternatively, highlight the entry and choose **Options** > **Display** or click the **Display** button. The relevant dialog box opens.
4. Click **OK** to close the dialog box.

The remaining sections in this chapter describe the results information that is provided for various event types.

Analyzing the Results of a Single-Property Check

A property check helps you to identify specific changes in the properties of 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 expected and actual results of a property check are displayed in the Property dialog box that you open from the Test Results window.



Property "enabled" for push_button "OK"

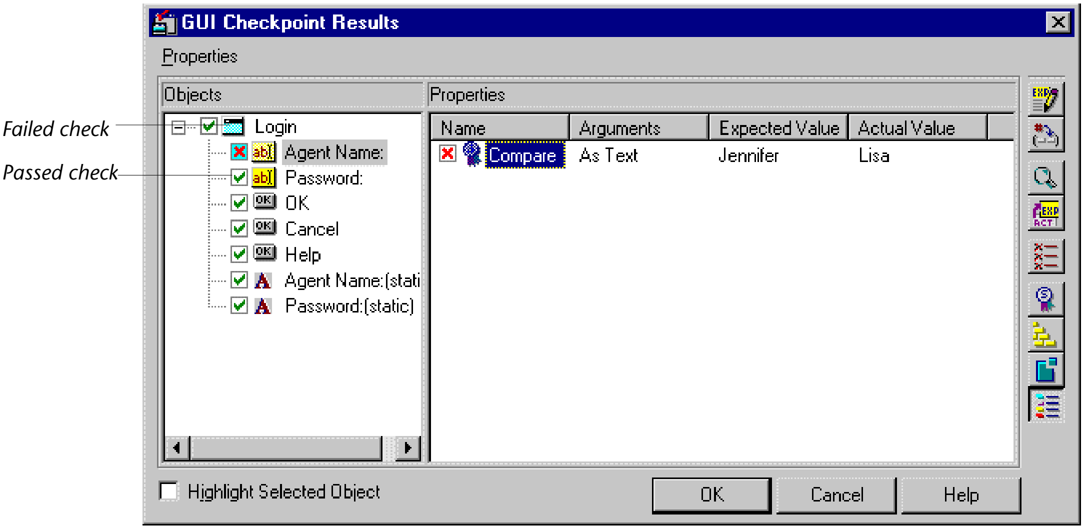
Expected value: 1

Actual value: 0

For more information, see Chapter 12, "Checking GUI Objects."

Analyzing 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.



GUI Checkpoint Results

Objects:

- ☑ Login
  - ☒ Agent Name:
  - ☑ Password:
  - ☑ OK
  - ☑ Cancel
  - ☑ Help
  - ☑ Agent Name:(stati
  - ☑ Password:(static)

Properties:

Name	Arguments	Expected Value	Actual Value
☒ Compare	As Text	Jennifer	Lisa

□ Highlight Selected Object

The dialog box lists every object checked and the types of checks performed. Each check is marked as either passed or failed and the expected and actual results are shown. If one or more objects fail, the entire GUI checkpoint is marked as failed in the test log.

You can update the expected value of a checkpoint, when working in the WinRunner report view. For additional information, see "Updating the Expected Results of a Checkpoint in the WinRunner Report View" on page 683. For a description of other options in this dialog box, see "Options in the GUI Checkpoint Results Dialog Box" on page 668.

For more information, see Chapter 12, "Checking GUI Objects."

Options in the GUI Checkpoint Results Dialog Box

The GUI Checkpoint Results dialog box includes the following options:

Button	Description
EXP	**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 219.
#	**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 213.
🔍	**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.
EXP ACT	**Update Expected Value** updates the expected value to the actual value. Note that this overwrites the saved expected value. This option is only available when working in the WinRunner report view.
✗	**Show Failures Only** displays only failed checks.
S	**Show Standard Properties Only** displays only standard properties.

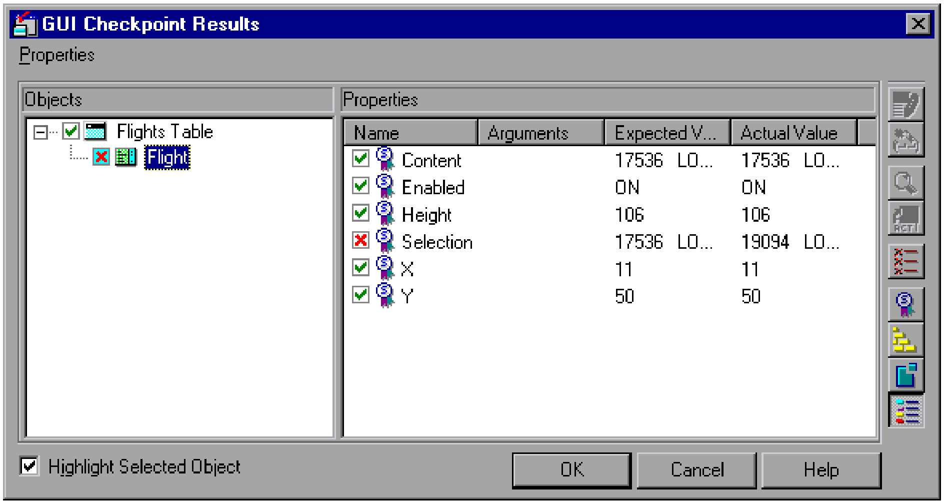
Button	Description
	Show Nonstandard Properties Only displays only nonstandard properties, such as Visual Basic, PowerBuilder, and ActiveX control properties.
	Show User Properties Only displays only user-defined property checks. To create user-defined property checks, refer to the *WinRunner Customization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

Analyzing 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 16, “Checking Table Contents.”

To display the results of a GUI checkpoint on table contents:

1 Open the GUI Checkpoint Results dialog box as described in "Viewing Checkpoint Results" on page 665.



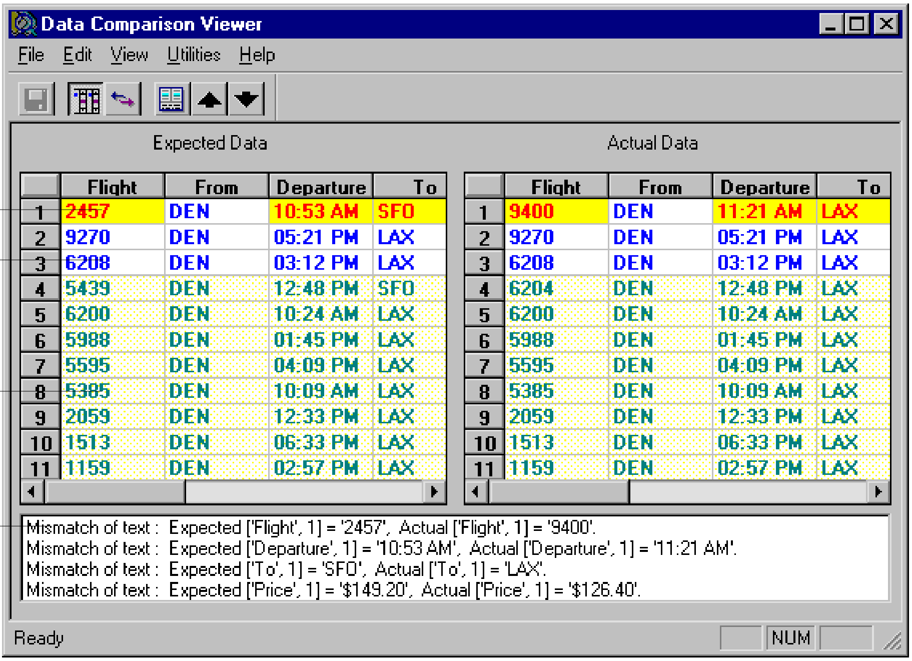
☑ Flights Table
☒ Flight

	Name	Arguments	Expected V...	Actual Value
☑	Content		17536 LO...	17536 LO...
☑	Enabled		ON	ON
☑	Height		106	106
☒	Selection		17536 LO...	19094 LO...
☑	X		11	11
☑	Y		50	50

☑ Highlight Selected Object

2 Highlight the table content checkpoint and click the **Display** button or double-click the table content checkpoint. In the example above, the Table content check is labeled **Flight**.

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.



Expected Data

	Flight	From	Departure	To
1	2457	DEN	10:53 AM	SFO
2	9270	DEN	05:21 PM	LAX
3	6208	DEN	03:12 PM	LAX
4	5439	DEN	12:48 PM	SFO
5	6200	DEN	10:24 AM	LAX
6	5988	DEN	01:45 PM	LAX
7	5595	DEN	04:09 PM	LAX
8	5385	DEN	10:09 AM	LAX
9	2059	DEN	12:33 PM	LAX
10	1513	DEN	06:33 PM	LAX
11	1159	DEN	02:57 PM	LAX

Actual Data

	Flight	From	Departure	To
1	9400	DEN	11:21 AM	LAX
2	9270	DEN	05:21 PM	LAX
3	6208	DEN	03:12 PM	LAX
4	6204	DEN	12:48 PM	LAX
5	6200	DEN	10:24 AM	LAX
6	5988	DEN	01:45 PM	LAX
7	5595	DEN	04:09 PM	LAX
8	5385	DEN	10:09 AM	LAX
9	2059	DEN	12:33 PM	LAX
10	1513	DEN	06:33 PM	LAX
11	1159	DEN	02:57 PM	LAX

```
Mismatch of text :  Expected ['Flight', 1] = '2457',  Actual ['Flight', 1] = '9400'.
Mismatch of text :  Expected ['Departure', 1] = '10:53 AM',  Actual ['Departure', 1] = '11:21 AM'.
Mismatch of text :  Expected ['To', 1] = 'SFO',  Actual ['To', 1] = 'LAX'.
Mismatch of text :  Expected ['Price', 1] = '$149.20',  Actual ['Price', 1] = '$126.40'.
```

Cell contains a mismatch.

Cell does not contain a mismatch.

Cell was not included in the comparison.

List of errors and mismatches.

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).

3. 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.

4. 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 tables scroll automatically to display it.

Note: When working in the WinRunner report view, 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 table content 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 304.

5. Choose **File** > **Exit** to close the Data Comparison Viewer.

Analyzing the Expected Results of a GUI Checkpoint on Table Contents

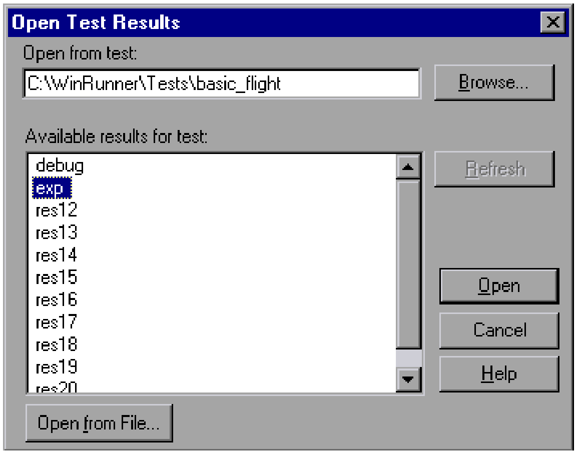
You can view the expected results of a GUI checkpoint on table contents either before or after you run your test. The expected results of a GUI checkpoint are displayed in the GUI Checkpoint Results dialog box, which you open from the Test Results window. When you view the expected results of a GUI checkpoint on table contents from the Test Results window, you must display the expected (“exp”).

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 16, “Checking Table Contents.”

To display the expected results of a GUI checkpoint on table contents:

1. Open the Test Results window and display the test for which you want to view expected results. For more information, see “Viewing Checkpoint Results” on page 665.
2. Display the expected results:

   ➤ In the Unified report view—Click the **Open** button or choose **File** > **Open**. The Open Test Results dialog box opens. Select **exp** and click **Open**.



Open Test Results

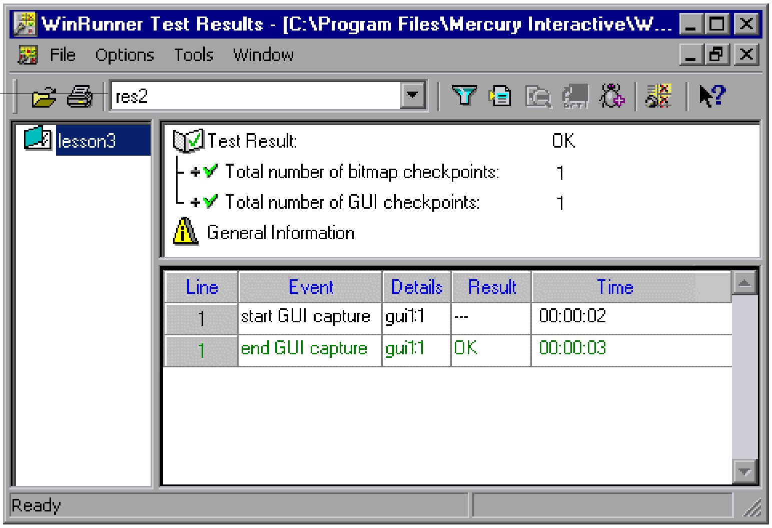
Open from test: C:\WinRunner\Tests\basic_flight

Available results for test:

debug
exp
res12
res13
res14
res15
res16
res17
res18
res19
res20

- ➤ In the WinRunner report view—Select **exp** in the Results location box.

Results location box



WinRunner Test Results - [C:\Program Files\Mercury Interactive\W...

Test Result: OK
Total number of bitmap checkpoints: 1
Total number of GUI checkpoints: 1
General Information

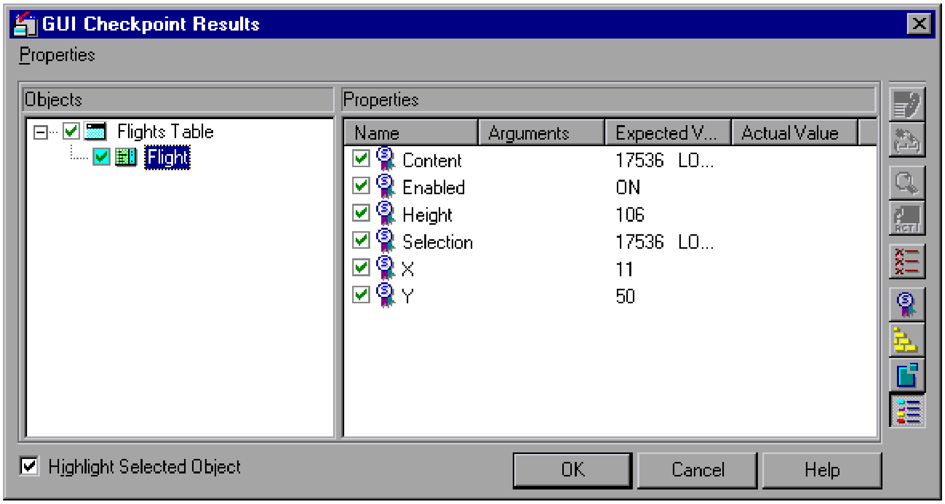
Line	Event	Details	Result	Time
1	start GUI capture	gui1:1	---	00:00:02
1	end GUI capture	gui1:1	OK	00:00:03

3 Display the expected results:

- ➤ In the Unified report view—Click the results tree node for the check you want to analyze. Basic details about the checkpoint are displayed in the **Event Summary** pane. In the Event Summary pane, click the **Show Event Details** link.

- ➤ In the WinRunner report view—Double-click an **End GUI capture** entry for a table check 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.



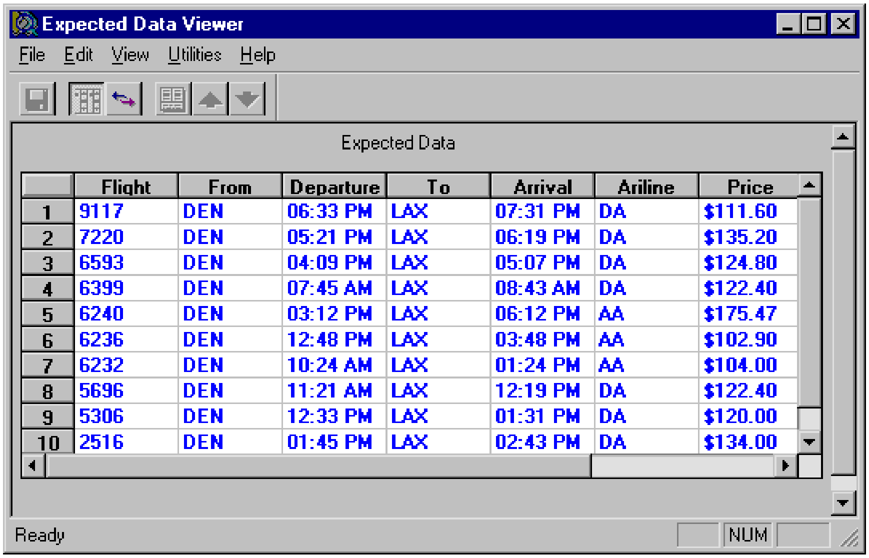
Name	Arguments	Expected V...	Actual Value
☑ Content		17536 LO...	
☑ Enabled		ON	
☑ Height		106	
☑ Selection		17536 LO...	
☑ X		11	
☑ Y		50	

Note: Since you are viewing the *expected* results of the GUI checkpoint, the *actual* values are not displayed.

---

4 Highlight the table content check and click the **Display** button, or double-click the table content check.

The Expected Data Viewer opens, displaying the expected results.



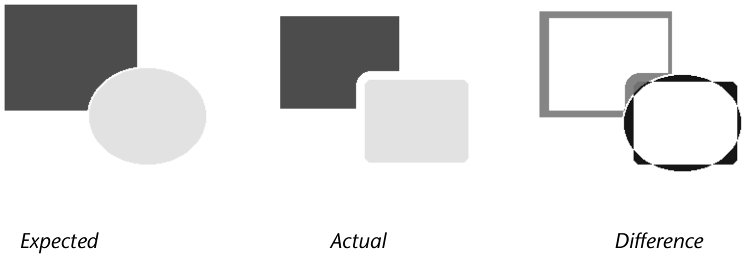
	Flight	From	Departure	To	Arrival	Ariline	Price
1	9117	DEN	06:33 PM	LAX	07:31 PM	DA	$111.60
2	7220	DEN	05:21 PM	LAX	06:19 PM	DA	$135.20
3	6593	DEN	04:09 PM	LAX	05:07 PM	DA	$124.80
4	6399	DEN	07:45 AM	LAX	08:43 AM	DA	$122.40
5	6240	DEN	03:12 PM	LAX	06:12 PM	AA	$175.47
6	6236	DEN	12:48 PM	LAX	03:48 PM	AA	$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: When working in the WinRunner report view, 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 304.

5 Choose **File** > **Exit** to close the Expected Data Viewer.

Analyzing 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 test run in Verify 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.

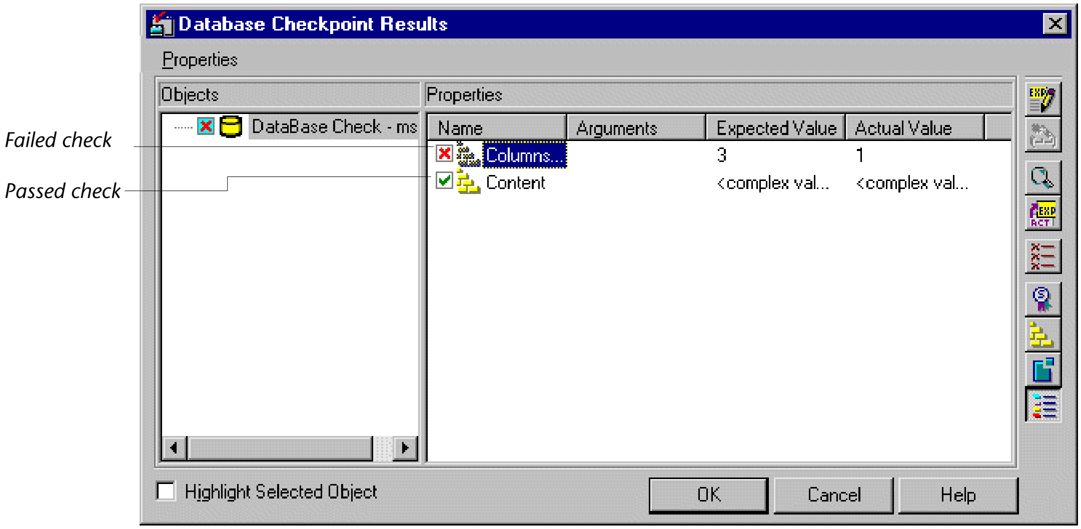


When viewing results in the WinRunner report view, you can control which types of bitmaps are displayed (expected, actual, difference) when you view the results of a bitmap checkpoint. To set the controls, choose **Options** > **Bitmap Controls** in the Test Results window.

Note: A bitmap checkpoint on identical bitmaps could fail if different display drivers are used when you create the checkpoint and when you run the test, because different display drivers may draw the same bitmap using slightly different color definitions. For more information, see “Handling Differences in Display Drivers,” on page 378.

Analyzing 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.



Name	Arguments	Expected Value	Actual Value
☒ Columns...		3	1
☑ Content		<complex val...	<complex val...

The dialog box displays the checked database and the types of checks performed. Each check is marked 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.

You can update the expected value of a checkpoint, when working in the WinRunner report view. For additional information on see "Updating the Expected Results of a Checkpoint in the WinRunner Report View" on page 683. For a description of other options in this dialog box, see "Options in the Database Checkpoint Results Dialog Box" on page 679.

Note: When working in the WinRunner report view, 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 343.

For more information, see Chapter 17, "Checking Databases."

Options in the Database Checkpoint Results Dialog Box

The Database Checkpoint Results dialog box includes the following options:

Button	Description
EXP	**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 333.
🔍	**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.
EXP ACT	**Update Expected Value** updates the expected value to the actual value. Note that this overwrites the saved expected value. This option is only available when working in the WinRunner report view.
✗	**Show Failures Only** displays only failed checks.
S	**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 All Properties displays all properties, including standard, nonstandard, and user-defined properties.

Analyzing 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 location 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 17, "Checking Databases."

To display the expected results of a content check in a database checkpoint:

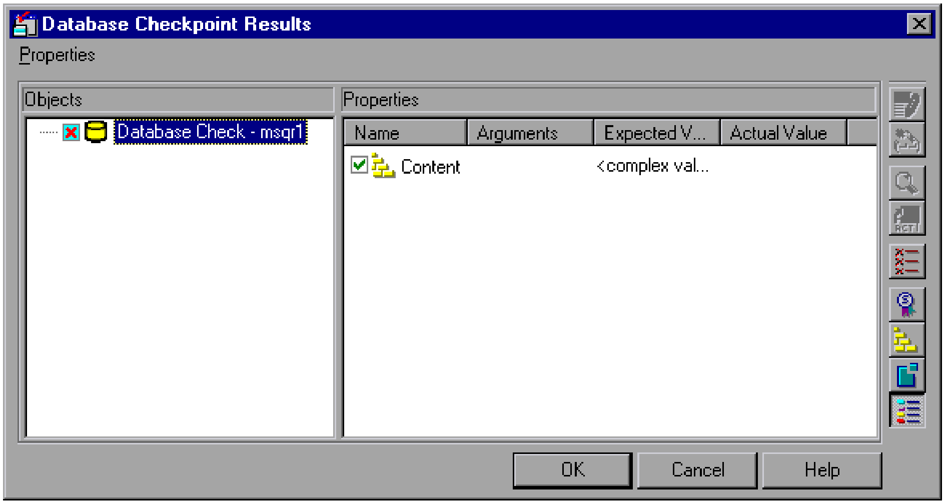
1. Open the Test Results window and display the test for which you want to add a defect. For more information, see "Viewing Checkpoint Results" on page 665.
2. Display the expected results:
   - ➤ In the Unified report view—Click the **Open** button or choose **File** > **Open**. The Open Test Results dialog box opens. Select **exp** and click **Open**.
   - ➤ In the WinRunner report view—Select **exp** in the Results location box.

Note that since you are viewing the *expected* results of a test, the total number of database checkpoints performed is listed as zero.

3 Display the expected results:

- ➤ In the Unified report view—Click the results tree node for the table check you want to analyze. Basic details about the checkpoint are displayed in the **Event Summary** pane. In the Event Summary pane, click the **Show Event Details** link.
- ➤ In the WinRunner report view—Double-click an **End GUI capture** entry for a table check 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.

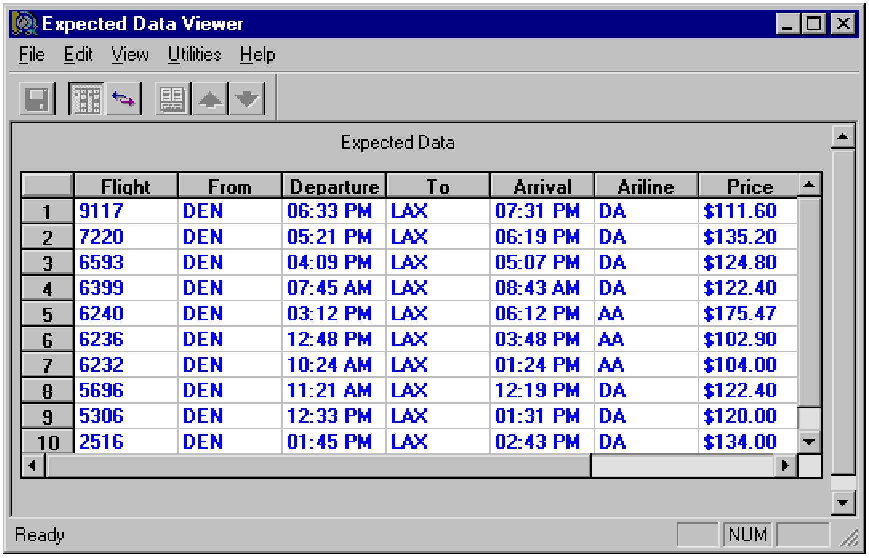


Name	Arguments	Expected V...	Actual Value
☑ Content		<complex val...	

Note that since you are viewing the *expected* results of the database checkpoint, the *actual* values are not displayed.

4 Highlight the database content check and click the **Display** button, or double-click the database 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: When working in the WinRunner report view, 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 343.

5 Choose **File** > **Exit** to close the Expected Data Viewer.

Updating the Expected Results of a Checkpoint in the WinRunner Report View

If a bitmap, GUI, or database checkpoint fails because the actual data is accurate but the expected data is incorrect, you can update the data in the expected results folder (**exp**) using the WinRunner report view.

For GUI and database checkpoints, you can update the results for the entire checkpoint, or update the results for a specific check within the checkpoint.

To update the expected results for an entire checkpoint:

1 In the WinRunner report view of the Test Results window, highlight a mismatched 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 checkpoint:

1 In the WinRunner report view of the Test Results window, double-click the checkpoint entry in the log, choose **Options** > **Display**, or click the **Display** button.

The relevant 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.

Viewing the Results of a File Comparison

If you used a **file_compare** statement in a test script to compare the contents of two files, you can view the results using the WDiff utility. This utility is accessed from the Test Results window.

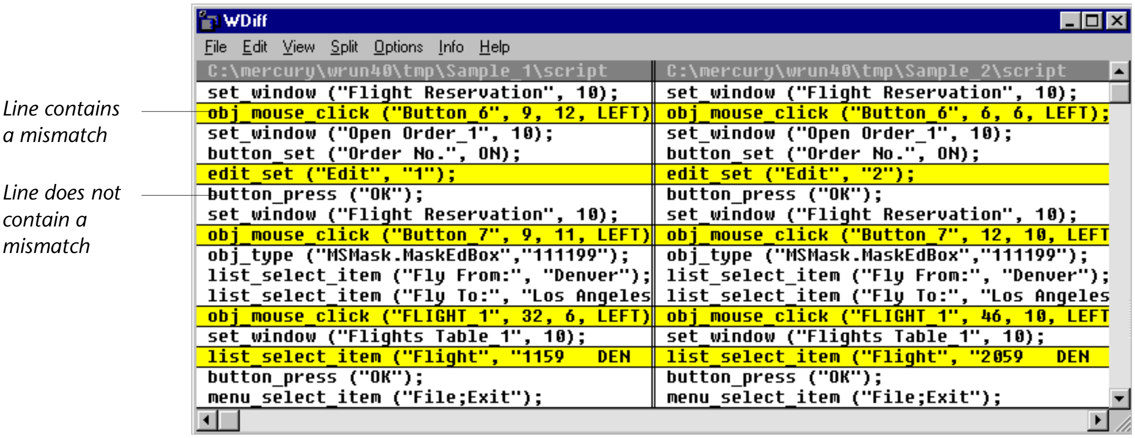
To view the results of a file comparison:

1. Open the Test Results window and display the test for which you want to view the file comparison results. For more information, see "Viewing Checkpoint Results" on page 665.
2. Display the file comparison:

   ➤ In the Unified report view—Click the results tree node for the file_compare event you want to analyze. Basic details about the checkpoint are displayed in the **Event Summary** pane. In the Event Summary pane, click the **Show Event Details** link.

   ➤ In the WinRunner report view—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.

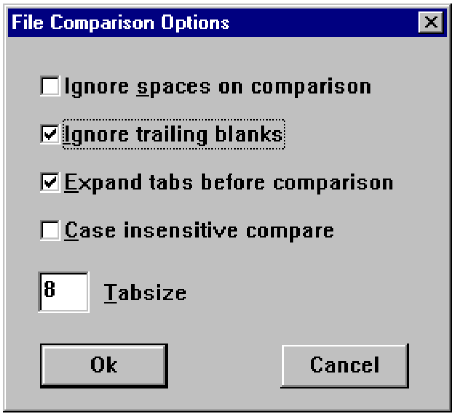


WDiff
File Edit View Split Options Info Help
Line contains a mismatch
Line does not contain a mismatch

```
C:\mercury\wrun40\tmp\Sample_1\script     | C:\mercury\wrun40\tmp\Sample_2\script
set_window ("Flight Reservation", 10);    | set_window ("Flight Reservation", 10);
obj_mouse_click ("Button_6", 9, 12, LEFT) | obj_mouse_click ("Button_6", 6, 6, LEFT);
set_window ("Open Order_1", 10);          | set_window ("Open Order_1", 10);
button_set ("Order No.", ON);             | button_set ("Order No.", ON);
edit_set ("Edit", "1");                   | edit_set ("Edit", "2");
button_press ("OK");                      | button_press ("OK");
set_window ("Flight Reservation", 10);    | set_window ("Flight Reservation", 10);
obj_mouse_click ("Button_7", 9, 11, LEFT) | obj_mouse_click ("Button_7", 12, 10, LEFT
obj_type ("MSMask.MaskEdBox","111199");   | obj_type ("MSMask.MaskEdBox","111199");
list_select_item ("Fly From:", "Denver"); | list_select_item ("Fly From:", "Denver");
list_select_item ("Fly To:", "Los Angeles | list_select_item ("Fly To:", "Los Angeles
obj_mouse_click ("FLIGHT_1", 32, 6, LEFT) | obj_mouse_click ("FLIGHT_1", 46, 10, LEFT
set_window ("Flights Table_1", 10);       | set_window ("Flights Table_1", 10);
list_select_item ("Flight", "1159   DEN   | list_select_item ("Flight", "2059   DEN
button_press ("OK");                      | button_press ("OK");
menu_select_item ("File;Exit");           | menu_select_item ("File;Exit");
```

The WDiff utility displays both files. Lines in the file that contain a mismatch are highlighted. The file defined in the first parameter of the **file_compare** statement is on the left side of the window.

- To see the next mismatch in a file, choose **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.
- To modify the way the actual and expected results are compared, choose **Options** > **File Comparison**. The File Comparison dialog box opens.



File Comparison Options

□ Ignore spaces on comparison

☑ Ignore trailing blanks

☑ Expand tabs before comparison

□ Case insensitive compare

8 Tabsize

Note that when you modify any of the options, the two files are read and compared again.

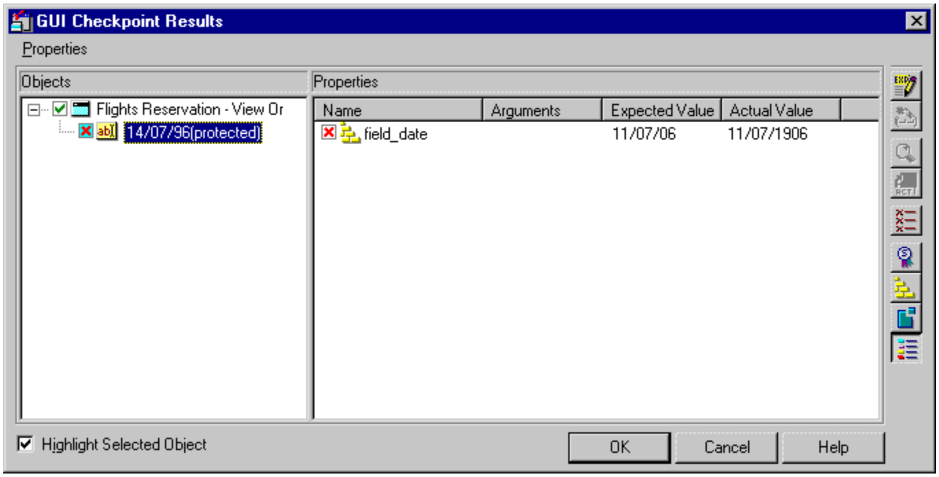
- **Ignore spaces on comparison:** Tab characters and spaces are ignored on comparison.
- **Ignore trailing blanks (default):** One or more blanks at the end of a line are ignored during the comparison.
- **Expand tabs before comparison (default):** Tab characters (hex 09) in the text are expanded to the number of spaces which are necessary to reach the next tab stop. The number of spaces between tab stops is specified in the **Tabsize** parameter. This **expand tabs before comparison** option will be ignored if the **Ignore spaces on comparison** option is selected at the same time.

- **Case insensitive compare:** Uppercase and lowercase is ignored during comparison of the files.
- **Tabsize:** The tabsize (number of spaces between tab stops) is selected between 1 and 19 spaces. The default size is 8 spaces. The option influences the file comparison if the **expand tabs before comparison** option is also set. Tabs are always expanded to the given number of spaces.

3 Choose **File** > **Exit** to close the WDiff Utility.

Viewing the Results of a GUI Checkpoint on a Date

You can check dates in GUI objects in your application. When you run your test, WinRunner compares the expected date with the actual date in the application. The results of a GUI checkpoint are displayed in the GUI Checkpoint Results dialog box that you open from the Test Results window.



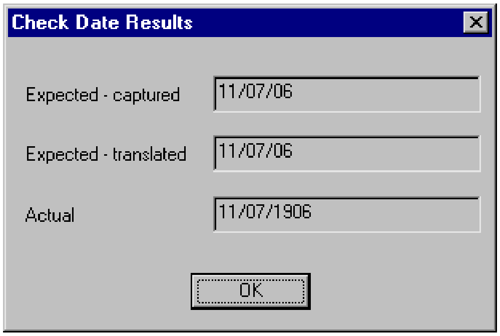
GUI Checkpoint Results

☑ Flights Reservation - View Or
☒ 14/07/96(protected)

Name	Arguments	Expected Value	Actual Value
☒ field_date		11/07/06	11/07/1906

☑ Highlight Selected Object

To view detailed information about a check on a date, double-click the check or click the **Compare Expected and Actual Values** button. The Check Date Results dialog box opens.



Check Date Results

Expected - captured: 11/07/06

Expected - translated: 11/07/06

Actual: 11/07/1906

The Check Date Results dialog box displays the original expected date, the expected date after aging and translation, and the actual date appearing in the object.

Reporting Defects Detected During a Test Run

Locating and repairing software defects efficiently is essential to the development process. Software developers, testers, and end users in all stages of the testing process can detect defects and add them to the defects project. Using Mercury Interactive's TestDirector Add Defect dialog box you can report design flaws in your application, and track data derived from defect reports.

For example, suppose you are testing a flight reservation application. You discover that errors occur when you try to order an airline ticket. You can 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, Web pages, text documents, and other files relevant to understanding and repairing the problem.

If a test run detects a defect in the application under test, you can report it directly from your Test Results window (when connected to a TestDirector project).

When you report a bug from the Test Results window, basic information about the test and the selected checkpoint (if applicable) is automatically included in the bug description.

Using the Add Defect Dialog Box

The Add Defect dialog box is a defect tracking component of TestDirector, Mercury Interactive's Web-based test management tool. You can report application defects directly to a TestDirector project. You can then track defects until the application's developers and software testers determine that they are resolved.

Setting Up the Add Defect Dialog Box

Before you can launch the Add Defect dialog box, you must ensure that TestDirector 7.2-8.0 is installed. You must also ensure that WinRunner is connected to a TestDirector server and project. The connection process has two stages. First, you connect WinRunner to the 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, test run information, and defects information for the application you are testing. For more information on connecting WinRunner to TestDirector, see "Connecting to TestDirector from the Test Results Window" on page 662.

For more information about installing TestDirector, refer to the *TestDirector Installation Guide*.

Reporting Defects with the Add Defect Dialog Box

When you are connected to TestDirector, you can report defects detected in your application directly from the WinRunner Test Results window.

To report a defect with the Add Defect dialog box:

1 If you are working in the WinRunner report view, connect to TestDirector from the main WinRunner window. For more information, see "Connecting to TestDirector from the Test Results Window" on page 662.

If you are working in the Unified report view, you need to connect to TestDirector from the Test Results window as described in step 4.

2. Open the Test Results window and display the test for which you want to add a defect. For more information, see “Viewing Checkpoint Results” on page 665.

3. If applicable, select the line in the Test Results that corresponds to the bug you want to report.

4. Open the Add Defect dialog box:

    - In the Unified report view—Click the **Add Defect** button or choose **Tools** > **Add Defect**. If the Test Results window is not yet connected to TestDirector, the TestDirector Connection dialog box opens. Connect to TestDirector as described in “Connecting to TestDirector from the Test Results Window” on page 662. When you are finished, click **Close** to close the TestDirector Connection dialog box and open the Add Defect Dialog box.

    - In the WinRunner report view—Click the **Report Bug** button or choose **Tools** > **Report Bug**.

    The Add Defect dialog box opens. Information about the selected line in the Test Results is included in the description.

5. Type a short description of the defect in **Summary**.

6. Enter information as appropriate in the rest of the defect text boxes. Note that you must enter information in all the text boxes with red labels.

7. Type a more in-depth description of the defect in the **Description** box.

    If you want to clear the data in the Add Defect dialog box, click the **Clear** button.

8. You can add an attachment to your defect report:

    - Click the **Attach File** button to attach a file to the defect.

    - Click the **Attach URL** button to attach a Web page to the defect.

    - Click the **Attach Screen Capture** button to capture an image and attach it to the defect.

9. Click the **Find Similar Defects** button to compare your defect to the existing defects in the TestDirector project. This lets you know if similar defect records already exist, and helps you to avoid duplicating them. If similar defects are found, they are displayed in the Similar Defects dialog box.

10. Click the **Submit** button to add the defect to the database. TestDirector assigns the new defect a Defect ID.
11. Click **Close**.

For more information on using the Add Defect dialog box, refer to the *TestDirector User's Guide*.

Reporting Defects During a Test Run

You can insert **tddb_add_defect** statements to your test to instruct WinRunner to add a defect to a TestDirector project based on conditions you define in your test script. Your statement can include data for the summary and description fields, as well as any other field name and value you specify.

For example, suppose your test begins by logging in to a flight reservation application. If the login is unsuccessful, you can report a defect that specifies the summary and description of the defect as well as the values for the **Detected by** and **Assigned to** fields.

Use the following syntax when inserting **tddb_add_defect** statements:

tddb_add_defect (*summary***,** *description***,** *defect_fields***);**

When entering defect fields, use the format:
"FieldName1=Value1;FieldName2=Value2;FieldNameN=ValueN".

Be sure to enter **field names** and not **field labels**. For example, use the field name **BG_DETECTED_BY** for the field label **Detected By**. For more information, refer to your TestDirector documentation.

If your test contains **tddb_add_defect** statements, confirm that you are connected to the appropriate TestDirector project before running your test.

35

Running Batch Tests

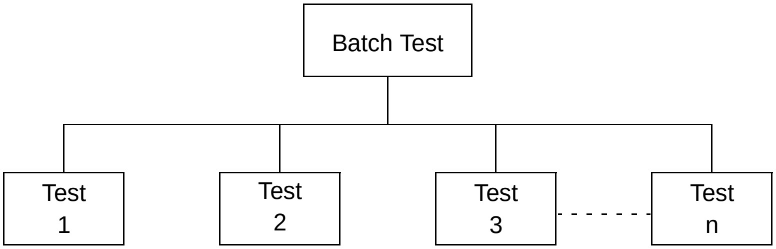
WinRunner enables you to execute a group of tests unattended. This can be particularly useful when you want to run a large group of tests overnight or at other off-peak hours.

This chapter describes:

- About Running Batch Tests
- Creating a Batch Test
- Running a Batch Test
- Storing Batch Test Results
- Viewing Batch Test Results

About Running Batch Tests

You can run a group of tests unattended by creating and executing a single batch test. A batch test is a test script that contains call statements to other tests. It opens and executes each test and saves the test results.



A batch test looks like a regular test that includes call statements. A test becomes a "batch test" when you select the **Run in batch mode** option in the **Run** category of the General Options dialog box before you execute the test.

When you run a test in Batch mode, WinRunner suppresses all messages that would ordinarily be displayed during the test run, such as a message reporting a bitmap mismatch. WinRunner also suppresses all **pause** statements and any halts in the test run resulting from run time errors.

By suppressing all messages, WinRunner can run a batch test unattended. This differs from a regular, interactive test run in which messages appear on the screen and prompt you to click a button in order to resume test execution. A batch test enables you to run tests overnight or during off-peak hours, so that you can save time while testing your application.

At each break during a test run—such as after a Step command, at a breakpoint, or at the end of a test, you can view the current chain of called tests in the Call Chain pane of the Debug Viewer window. For more information, see "Viewing the Call Chain" on page 579

When a batch test run is completed, you can view the results in the Test Results window. The window displays the results of all the major events that occurred during the run.

Note that you can also run a group of tests from the command line. For details, see Chapter 36, "Running Tests from the Command Line."

Creating a Batch Test

A batch test is a test script that calls other tests. You program a batch test by typing call statements directly into the test window and selecting the **Run in batch mode** option in the **Run** category of the General Options dialog box before you execute the test.

A batch test may include programming elements such as loops and decision-making statements. Loops enable a batch test to run called tests a specified number of times. Decision-making statements such as *if/else* and *switch* condition test execution on the results of a test called previously by the same batch script. See Chapter 26, “Enhancing Your Test Scripts with Programming,” for more information.

For example, the following batch test executes three tests in succession, then loops back and calls the tests again. The loop specifies that the batch test should call the tests ten times.

```
for (i=0; i<10; i++)
    {
    call "c:\\pbtests\\open" ();
    call "c:\\pbtests\\setup" ();
    call "c:\\pbtests\\save" ();
    }
```

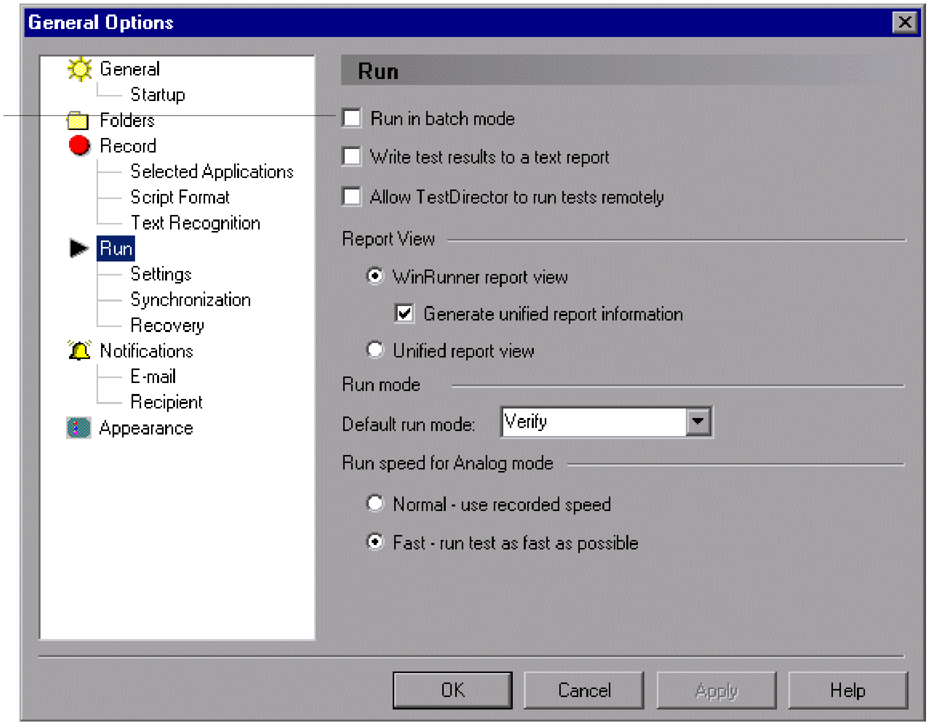
To enable a batch test:

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

The General Options dialog box opens.

2 Click the **Run** category.

3 Select the **Run in batch mode** check box.



Run in batch mode

Run

☐ Run in batch mode

☐ Write test results to a text report

☐ Allow TestDirector to run tests remotely

Report View

● WinRunner report view

☑ Generate unified report information

○ Unified report view

Run mode

Default run mode: Verify

Run speed for Analog mode

○ Normal - use recorded speed

● Fast - run test as fast as possible

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

For more information on setting the batch option in the General Options dialog box, see Chapter 41, "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 **Test** > **Run from Top**. See Chapter 33, "Understanding Test Runs," for more information.

When you run a batch test, WinRunner opens and executes each called test. All messages are suppressed so that the tests are run without interruption. If you run the batch test in **Verify** mode, the current test results are compared to the expected test results saved earlier. If you are running the batch test in order to update expected results, new expected results are created in the expected results folder for each test. See "Storing Batch Test Results" below for more information. When the batch test run is completed, you can view the test results in the Test Results window.

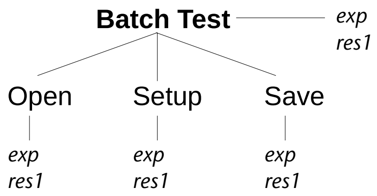
Note that if your tests contain TSL **texit** statements, WinRunner interprets these statements differently for a batch test run than for a regular test run. During a regular test run, **texit** terminates test execution. During a batch test run, **texit** halts execution of the current test only and control is returned to the batch test.

Storing Batch Test Results

When you run a regular, interactive test, results are stored in a subfolder under the test. If **Run in batch mode** is selected in the **Run** category of the General Options dialog box, then WinRunner saves the results for each (top-level) called test separately in a subfolder under the called test. Additionally, a subfolder is also created for the batch test that contains the results of the entire batch test run, including all called tests.

For example, suppose you create three tests: *Open*, *Setup*, and *Save*. For each test, expected results are saved in an *exp* subfolder under the test folder. Suppose you also create a batch test that calls the three tests. Before running the batch test in **Verify** mode, you instruct WinRunner to save the results in a subfolder of the calling test called *res1*. When the batch test is run, it compares the current test results to the expected results saved earlier.

Under each test folder, WinRunner creates a subfolder called *res1* in which it saves the verification results for the test. A *res1* folder is also created under the batch test to contain the overall verification results for the entire run.



If you run the batch test in **Update** mode in order to update expected results, WinRunner overwrites the expected results in the *exp* subfolder for each test and for the batch test.

Notes:

If a called test already had a folder called *res1*, when the batch run results create folders under each test called *res1*, those results overwrite the previous *res1* results in the called test's folder.

If you run the batch test without selecting the **Run in batch mode** check box (**Tools** > **General Options** > **Run**), WinRunner saves results only in the subfolder for the batch test. This can cause problems at a later stage if you choose to run the called tests separately, since WinRunner will not know where to look for the previously saved expected and verification results. When working in unified report mode, all batch run results are saved in a single results folder under the main test's folder.

If a called test calls additional tests, then those results are saved only in the results folder of the test that called it. For example, suppose test A calls tests B and C, test B calls tests D, and E, and test E calls test Z, then when running in batch mode, the results of test B are stored under test A and also under test B, but the results of tests C,D,E, and Z are all stored only under the main batch test (A) and also under the top-level called test (B).

Viewing Batch Test Results

When a batch test run is completed, you can view information about the events that occurred during the run in the Test Results window. If one of the called tests fails, the batch test is marked as failed.

The Test Results window lists all the events that occurred during the batch test run. Each time a test is called, a *call_test* entry is listed. The details of the *call_test* entry indicate whether the **call** statement was successful. Note that even though a **call** statement is successful, the called test itself may fail, based on the usual criteria for a failed tests . You can set criteria for a failed test in the **Run** > **Settings** category of the General Options dialog box. For additional information, see Chapter 41, "Setting Global Testing Options."

To view the results of the called test, double-click the *call_test* entry. For more information on viewing test results in the Test Results window, see Chapter 34, "Analyzing Test Results."

36

Running Tests from the Command Line

You can run tests directly from the Windows command line.

This chapter describes:

- About Running Tests from the Command Line
- Using the Windows Command Line
- Command Line Options

About Running Tests from the Command Line

You can use the Windows Run command to start WinRunner and run a test according to predefined options. You can also save your startup options by creating a custom WinRunner shortcut. Then, to start WinRunner with the startup options, you simply double-click the icon.

Using the command line, you can:

- start your application
- start WinRunner
- load the relevant tests
- run the tests
- specify test options
- specify the results directories for the test

Most of the functional options that you can set within WinRunner can also be set from the command line. These include test run options and the directories in which test results are stored. You can also specify a *custom.ini* file that contains these and other environment variables and system parameters.

For example, the following command starts WinRunner, loads a batch test, and runs the test in verify mode:

```
C:\Program Files\Mercury Interactive\WinRunner\WRUN.EXE -t
c:\batch\newclock -batch on -verify -run_minimized -dont_quit -run
```

The test *newclock* is loaded and then executed in batch mode with WinRunner minimized. WinRunner remains open after the test run is completed.

Note: You can use AT commands (specifically the SU.EXE command) with WinRunner. AT commands are part of the Microsoft Windows NT operating system. You can find information on AT commands in the NT Resource Kit. This enables running completely automated scripts, without user intervention.

Using the Windows Command Line

You can use the Windows command line to start WinRunner with predefined options. If you plan to use the same set of options each time you start WinRunner, you can create a custom WinRunner shortcut.

Starting WinRunner from the Command Line

This procedure describes how to start WinRunner from the command line.

To start WinRunner from the Run command:

1 On the Windows **Start** menu, choose **Run**. The Run dialog box opens.

2 Type in the path of your WinRunner *wrun.exe* file, and then type in any command line options you want to use.

3 Click **OK** to close the dialog box and start WinRunner.

Note: If you add command line options to a path containing spaces, you must specify the path of the wrun.exe within quotes, for example:

```
"D:\Program Files\Mercury Interactive\WinRunner\arch\wrun.exe" -addins WebTest
```

Adding a Custom WinRunner Shortcut

You can make the options you defined permanent by creating a custom WinRunner shortcut.

To add a custom WinRunner shortcut:

1 Create a shortcut for your *wrun.exe* file in Windows Explorer or My Computer.

2 Click the right mouse button on the shortcut and choose **Properties**.

3 Click the **Shortcut** tab.

4 In the **Target** box, type in any command line options you want to use after the path of your WinRunner *wrun.exe* file.

5 Click **OK**.

Command Line Options

Following is a description of each command line option.

-addins *list of add-ins to load*

Instructs WinRunner to load the specified add-ins. In the list, separate the add-ins by commas (without spaces). This can be used in conjunction with the **-addins_select_timeout** command line option.

(Formerly **-addons.**)

Note: All installed add-ins are listed in the registry under: *HKEY_LOCAL_MACHINE\SOFTWARE\Mercury Interactive\WinRunner\CurrentVersion\Installed Components\.*

Use the syntax (spelling) displayed in the key names under this branch when specifying the add-ins to load. The names of the add-ins are not case sensitive.

For example, the following line will load the four add-ins that are included with WinRunner:

```
<WinRunner folder>\arch\wrun.exe -addins ActiveX,pb,vb,WebTest
```

-addins_select_timeout *timeout*

Instructs WinRunner to wait the specified time (in seconds) before closing the **Add-In Manager** dialog box when starting WinRunner. When the timeout is zero, the dialog box is not displayed. This can be used in conjunction with the **-addins** command line option.

(Formerly **-addons_select_timeout.**)

-animate

Instructs WinRunner to execute and run the loaded test, while the execution arrow displays the line of the test being run.

-app *path*

Runs the specified application before running WinRunner. This can be used in conjunction with the **-app_params, -app_open_win,** and **-WR_wait_time** command line options.

Note that you can also define a startup application in the **Run** tab of the Test Properties dialog box. For more information, see Chapter 40, "Setting Properties for a Single Test."

-app_params *param1[,param2,...,paramN]*

Passes the specified parameters to the application specified in **-app.**

Note: You can only use this command line option when you also use the **-app** command line option.

-app_open_win *setting*

Determines how the application window appears when it opens.

The following are the possible values for *setting*:

Option	Description
SW_HIDE	Hides the window and activates another window.
SW_SHOWNORMAL	Activates and displays a window. If the window is minimized or maximized, Windows restores it to its original size and position. Specify this flag when displaying the window for the first time.
SW_SHOWMINIMIZED	Activates the window and displays it as a minimized window.
SW_SHOWMAXIMIZED	Activates the window and displays it as a maximized window.
SW_SHOWNOACTIVATE	Displays a window in its most recent size and position. The active window remains active.

Option	Description
SW_SHOW	Activates the window and displays it in its current size and position.
SW_MINIMIZE	Maximizes the specified window and activates the next top-level window in the z-order.
SW_SHOWMINNOACTIVE	Displays the window as a minimized window. The active window remains active.
SW_SHOWNA	Displays the window in its current state. The active window remains active.
SW_RESTORE	Activates and displays the window. If the window is minimized or maximized, Windows restores it to its original size and position. Specify this flag when restoring a minimized window.

Note: You can only use this command line option when you also use the **-app** command line option.

-auto_load {on | off}

Activates or deactivates automatic loading of the temporary GUI map file.

(Default = **on**)

-auto_load_dir *path*

Determines the folder in which the temporary GUI map file (*temp.gui*) resides. This option is applicable only when auto load is on.

(Default = **M_Home\dat**)

-batch {on | off}

Runs the loaded test in Batch mode.

(Default = **off**)

You can also set this option using the **Run in batch mode** check box in the **Run** category of the General Options dialog box as described in "Setting Test Run Options" on page 793.

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 44, "Setting Testing Options from a Test Script."

Tip: To ensure that the test run does not pause to display error messages, use the **-batch** option in conjunction with the **-verify** option. For more information on the **-verify** option, see page 717.

-beep {on | off}

Activates or deactivates the WinRunner system beep.

You can also set this option using the corresponding **Beep when checking a window** check box in the **Run** > **Settings** category of the General Options dialog box, described in "Setting Run Setting Options" on page 797.

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 44, "Setting Testing Options from a Test Script."

-capture_bitmap {on | off }

Determines whether WinRunner captures a bitmap whenever a checkpoint fails. When this option is on (1), WinRunner uses the settings from the **Run** > **Settings** category of the General Options dialog box to determine the captured area for the bitmaps.

(Default = **off**)

You can also set this option using the **Capture bitmap on verification failure** check box in the **Run** > **Settings** category of the General Options dialog box, as described in "Setting Run Setting Options" on page 797.

Note that you can use the **setvar** and **getvar** functions to set and retrieve the value of the corresponding *capture_bitmap* testing option from within a test script, as described in Chapter 44, "Setting Testing Options from a Test Script."

-create_text_report {on | off}

Instructs WinRunner to write test results to a text report, *report.txt*, which is saved in the results folder.

-create_unirep_info {on | off}

Generates the necessary information for creating a Unified Report (when WinRunner report view is selected) so that you can choose to view the Unified Report of your tests at a later time.

(Default = **on**)

You can also set this option using the corresponding **Create unified report information** option in the **Run** category of the General Options dialog box as described in "Setting Test Run Options" on page 793.

-cs_fail {on | off}

Determines whether WinRunner fails a test when Context Sensitive errors occur. A Context Sensitive error is the failure of a Context Sensitive statement during a test. Context Sensitive errors are often due to WinRunner's failure to identify a GUI object.

For example, a Context Sensitive error will occur if you run a test containing a **set_window** statement with the name of a non-existent window. Context Sensitive errors can also occur when window names are ambiguous. For information about Context Sensitive functions, refer to the *TSL Reference*.

(Default = **off**)

You can also set this option using the corresponding **Fail test when Context Sensitive errors occur** check box in the **Run** > **Settings** category of the General Options dialog box, described in "Setting Run Setting Options" on page 797

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 44, "Setting Testing Options from a Test Script."

-cs_run_delay *non-negative integer*

Sets the time (in milliseconds) that WinRunner waits between executing Context Sensitive statements when running a test.

(Default = **0** [milliseconds])

You can also set this option using the corresponding **Delay between execution of CS statements** box in the **Run** > **Synchronization** category of the General Options dialog box, described in "Setting Run Synchronization Options" on page 802.

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 44, "Setting Testing Options from a Test Script."

-def_replay_mode {verify | debug | update}

Sets the run mode that is used by default for all tests .

Possible values:

- **Update—**Used to update the expected results of a test or to create a new expected results folder.
- **Verify—**Used to check your application.
- **Debug—**Used to help you identify bugs in a test script.

(Default = **Verify**)

You can also set this option using the **Default run mode** option in the **Run** category of the General Options dialog box as described in "Setting Test Run Options" on page 793.

-delay_msec *non-negative integer*

Directs WinRunner to determine whether a window or object is stable before capturing it for a bitmap checkpoint or synchronization point. It defines the time (in milliseconds) that WinRunner waits between consecutive samplings of the screen. If two consecutive checks produce the same results, WinRunner captures the window or object. (Formerly **-delay**, which was measured in seconds.)

(Default = **1000** [milliseconds])

(Formerly **-delay**.)

Note: This parameter is accurate to within 20-30 milliseconds.

You can also set this option using the corresponding **Delay for window synchronization** box in the **Run** > **Synchronization** category of the General Options dialog box, described in "Setting Run Synchronization Options" on page 802.

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 44, "Setting Testing Options from a Test Script."

-dont_connect

If the **Reconnect on startup** option is selected in the Connection to Test Director dialog box, this command line enables you to open WinRunner without connecting to Test Director.

To disable the **Reconnect on startup** option, select **Tools** > **TestDirector Connection** and clear the **Reconnect on startup c**heck box as described in Chapter 36, "Running Tests from the Command Line".

-dont_quit

Instructs WinRunner not to close after completing the test.

-dont_show_welcome

Instructs WinRunner not to display the Welcome window when starting WinRunner.

-email_service

Determines whether WinRunner activates the e-mail sending options including the e-mail notifications for checkpoint failures, test failures, and test completed reports as well as any **email_send_msg** statements in the test.

(Default = **off**)

You can also set this option using the corresponding Activate e-mail service check box in the **Notifications** > **E-mail** category of the General Options dialog box as described in "Setting E-mail Notification Options" on page 810.

Note that you can use the **setvar** and **getvar** functions to set and retrieve the value of the corresponding **email_service** testing option from within a test script, as described in Chapter 44, "Setting Testing Options from a Test Script."

-exp *expected results folder name*

Designates a name for the subfolder in which expected results are stored. In a verification run, specifies the set of expected results used as the basis for the verification comparison.

(Default = **exp**)

You can also view this setting using the corresponding **Expected results folder** box in the **Current Test** tab of the Test Properties dialog box, described in "Reviewing Current Test Settings" on page 757.

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 44, "Setting Testing Options from a Test Script."

-fast_replay {on | off}

Sets the speed of the test run for tests recorded in analog mode. **on** sets tests to run as fast as possible and **off** sets tests to run at the speed at which they were recorded.

Note that you can also specify the analog run speed using the **Run speed for Analog mode** option in the **Run** category of the General Options dialog box as described in "Setting Test Run Options" on page 793.

(Default = **on**)

-f *file name*

Specifies a text file containing command line options. The options can appear on the same line, or each on a separate line. This option enables you to circumvent the restriction on the number of characters that can be typed into the Target text box in the **Shortcut** tab of the Windows Properties dialog box.

Note: If a command line option appears both in the command line and in the file, WinRunner uses the settings of the option in the file.

-fontgrp *group name*

Specifies the active font group when WinRunner is started.

You can also set this option using the corresponding **Font group** box in the **Record** > **Text Recognition** category of the General Options dialog box, described in "Setting Text Recognition Options" on page 790.

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 44, "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])

You can also set this option using the corresponding **Threshold for difference between bitmaps** box in the **Run** > **Settings** category of the General Options dialog box, described in "Setting Run Setting Options" on page 797.

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 44, "Setting Testing Options from a Test Script."

-mismatch_break {on | off}

Activates or deactivates Break when Verification Fails before a verification run. The functionality of Break when Verification Fails is different than when running a test interactively: In an interactive run, the test is paused; For a test started from the command line, the first occurrence of a comparison mismatch terminates the test run.

Break when Verification Fails determines whether WinRunner pauses the test run and displays a message whenever verification fails or whenever any message is generated as a result of a Context Sensitive statement during a test that is run in Verify mode.

For example, if a **set_window** statement is missing from a test script, WinRunner cannot find the specified window. If this option is on, WinRunner pauses the test and opens the Run wizard to enable the user to locate the window. If this option is off, WinRunner reports an error in the Test Results window and proceeds to run the next statement in the test script.

(Default = **on**)

You can also set this option using the corresponding **Break when verification fails** check box in the **Run** > **Settings** category of the General Options dialog box, described in "Setting Run Setting Options" on page 797.

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 44, "Setting Testing Options from a Test Script."

-rec_item_name {0 | 1}

Determines whether WinRunner records non-unique ListBox and ComboBox items by name or by index.

(Default = **0**)

You can also set this option using the corresponding **Record non-unique list items by name** check box in the **Record** category of the General Options dialog box, described in "Setting Recording Options" on page 778.

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 44, "Setting Testing Options from a Test Script."

-run

Instructs WinRunner to run the loaded test. To load a test into the WinRunner window, use the **-t** command line option.

-run_minimized

Instructs WinRunner to open and run tests with WinRunner and the test minimized to an icon. Note that specifying this option does not itself run tests: use the **-t** command line option to load a test and the **-run** command line option to run the loaded test.

-search_path *path*

Defines the directories to be searched for tests to be opened and/or called. The search path is given as a string.

(Default = **startup folder** and **installation folder\lib**)

You can also set this option using the corresponding **Search path for called tests** box in the **Folders** category of the General Options dialog box, described in "Setting Folder Options" on page 775.

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 44, “Setting Testing Options from a Test Script.”

-single_prop_check_fail {on | off}

Fails a test run when **_check_info** statements fail. It also writes an event to the Test Results window for these statements. (You can create **_check_info** statements using the **Insert** > **GUI Checkpoint** > **For Single Property** command.)

You can use this option with the **setvar** and **getvar** functions.

(Default = **on**)

For information about the **check_info** functions, refer to the *TSL Reference*.

You can also set this option using the corresponding **Fail test when single property check fails** option in the **Run** > **Settings** category of the General Options dialog box, described in “Setting Run Setting Options” on page 797.

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 44, “Setting Testing Options from a Test Script.”

-speed {normal | fast}

Sets the speed for the execution of the loaded test.

(Default = **fast**)

You can also set this option using the corresponding **Run Speed for Analog Mode** option in the **Run** category of the General Options dialog box, described in “Setting Test Run Options” on page 793.

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 44, “Setting Testing Options from a Test Script.”

(Formerly **-run_speed.**)

-start_minimized {on | off}

Indicates whether WinRunner opens in minimized mode.

(Default = **off**)

-t *test name*

Specifies the name of the test to be loaded in the WinRunner window. This can be the name of a test stored in a folder specified in the search path or the full pathname of any test stored in your system.

-td_connection {on | off}

Activates WinRunner's connection to TestDirector when set to **on**.

(Default = **off**)

(Formerly **-test_director.**)

Note that you can connect to TestDirector from the TestDirector Connection dialog box, which you open by choosing **Tools** > **TestDirector Connection**. For more information about connecting to TestDirector, see Chapter 48, "Managing the Testing Process."

Note: If you select the "Reconnect on startup" option in the Connection to Test Director dialog box, setting **-td_connection** to off will not prevent the connection to TestDirector. To prevent the connection to TestDirector in this situation, use the **-dont_connect** command. For more information, see "-dont_connect," on page 708.

-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 44, "Setting Testing Options from a Test Script."

(Formerly **-cycle.**)

-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 44, "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 TestDirector Connection dialog box, which you open by choosing **Tools** > **TestDirector Connection**. For more information, see Chapter 48, "Managing the Testing Process."

(Formerly **-database.**)

-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 TestDirector Connection dialog box, which you open by choosing **Tools** > **TestDirector Connection**. For more information about connecting to TestDirector, see Chapter 48, "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 44, "Setting Testing Options from a Test Script."

In order to connect to the server, use the **td_connection** option.

(Formerly **-td_server.**)

-td_user_name *user name*

Specifies the name of the user who is currently executing a test cycle.

Note that you can use the corresponding *td_user_name* testing option to specify the user, as described in Chapter 44, "Setting Testing Options from a Test Script."

Note that you can specify the user name when you connect to TestDirector from the TestDirector Connection dialog box, which you open by choosing **Tools** > **TestDirector Connection**. For more information about connecting to TestDirector, see Chapter 48, "Managing the Testing Process."

(Formerly **-user_name** or **user.**)

-timeout_msec *non-negative integer*

Sets the global timeout (in milliseconds) used by WinRunner when executing checkpoints and Context Sensitive statements. This value is added to the *time* parameter embedded in GUI checkpoint or synchronization point statements to determine the maximum amount of time that WinRunner searches for the specified window or object. (Formerly *timeout*, which was measured in seconds.)

(Default = **10,000** [milliseconds])

(Formerly **-timeout**.)

Note: This option is accurate to within 20-30 milliseconds.

You can also set this option using the corresponding **Timeout for checkpoints and CS statements** box in the **Run** > **Settings** category of the General Options dialog box, described in "Setting Run Setting Options" on page 797.

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 44, "Setting Testing Options from a Test Script."

-tslinit_exp *expected results folder*

Directs WinRunner to the expected folder to be used when the *tslinit* script is running.

-update_ini

Saves changes to configuration made during a WinRunner session when the *wrun.ini* file is specified by the **-ini** command line option.

Note: You can only use this command line option when you also use the **-ini** command line option.

-verify *verification results folder name*

Specifies that the test is to be run in **Verify** mode and designates the name of the subfolder in which the test results are stored.

-WR_wait_time *non-negative integer*

Specifies the number of milliseconds to wait between invoking the application and starting WinRunner.

(Default = **0** [milliseconds])

You can also set this option using the **Run test after** box in the **Run** tab of the Test Properties dialog box, described in Chapter 40, “Setting Properties for a Single Test.”

Note: You can only use this command line option when you also use the **-app** command line option.

Part VI

Debugging Tests

37

Controlling Your Test Run

Controlling the test run can help you to identify and eliminate defects in your test scripts.

This chapter describes:

- About Controlling Your Test Run
- Running a Single Line of a Test Script
- Running a Section of a Test Script
- Pausing a Test Run

About Controlling Your Test Run

After you create a test script you should check that it runs smoothly, without errors in syntax or logic. In order to detect and isolate defects in a script, you can use the Step and Pause commands to control test execution.

The following Step commands are available:

- The Step command runs a single line of a test script.
- The Step Into command calls and displays another test or user-defined function.
- The Step Out command—used in conjunction with Step Into—completes the execution of a called test or user-defined function.
- The *S*tep to Cursor command runs a selected section of a test script.

In addition, you can use the Pause command or the **pause** function to temporarily suspend the test run.

You can also control the test run by setting breakpoints. A breakpoint pauses a test run at a pre-determined point, enabling you to examine the effects of the test on your application. You can view all breakpoints in the Breakpoints List pane of the Debug Viewer. For more information, see Chapter 38, "Using Breakpoints."

To help you debug your tests, WinRunner enables you to monitor variables in a test script. You define the variables you want to monitor in a Watch List. As the test runs, you can view the values that are assigned to the variables. You can view the current values of monitored variables in the Watch List pane of the Debug Viewer. For more information, see Chapter 39, "Monitoring Variables."

You can use the call chain to follow and navigate the test flow. At each break during a test run—such as after a Step command, at a breakpoint, or at the end of a test, you can view the current chain of called tests and functions in the Call Chain pane of the Debug Viewer. For more information, see Chapter 28, "Calling Tests."

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 33, "Understanding Test Runs."

Running a Single Line of a Test Script

You can run a single line of a test script using the Step, Step Into, and Step Out commands.

Step

Choose the **Step** command or click the corresponding **Step** button to execute only the current line of the active test script—the line marked by the execution arrow.

When the current line calls another test or a user-defined function, the called test or function is executed in its entirety but the called test script is not displayed in the WinRunner window. If you are using a startup application or startup function, it is also executed.

Step Into

Choose the **Step Into** command or click the corresponding **Step Into** button to execute only the current line of the active test script. However, in contrast to Step, if the current line of the executed test calls another test or a user-defined function in compiled mode:

- The test script of the called test or function is displayed in the WinRunner window.
- Startup application and function settings (Test Properties dialog box, **Run** tab) are not implemented.
- Use Step or Step Out to continue running the called test.

Step Out

You use the **Step Out** command only after entering a test or a user-defined function using Step Into. Step Out executes to the end of the called test or user-defined function, returns to the calling test, and then pauses the test run.

Running a Section of a Test Script

You can execute a selected section of a test script using the Step to Cursor command.

To use the Step to Cursor command:

1. Move the execution arrow to the line in the test script from which you want to begin test execution. To move the arrow, click inside the margin next to the desired line in the test script.
2. Click inside the test script to move the cursor to the line where you want test execution to stop.

3 Choose **Debug >Step to Cursor** or press the STEP TO CURSOR softkey. WinRunner runs the test up to the line marked by the insertion point.

Pausing a Test Run

You can temporarily suspend a test run by choosing the Pause command or by adding a **pause** statement to your test script.

Pause Command

You can suspend the running of a test by choosing **Test** > **Pause**, clicking the **Pause** button, or pressing the PAUSE softkey. A paused test stops running when all previously interpreted TSL statements have been executed. Unlike the **Stop** command, **Pause** does not initialize test variables and arrays.

To resume running of a paused test, choose the appropriate Run command on the **Test** menu. The test run continues from the point that you invoked the Pause command, or from the execution arrow if you moved it while the test was suspended.

The pause Function

When WinRunner processes a **pause** statement in a test script, test execution halts and a message box is displayed. If the **pause** statement includes an expression, the result of the expression appears in the message box. The syntax of the **pause** function is:

pause ([*expression*] **);**

In the following example, **pause** suspends the test run and displays the time that elapsed between two points.

```
t1=get_time();
t2=get_time();
pause ("Time elapsed" is & t2-t1);
```

Note: The **pause** statement is ignored by WinRunner when running tests in batch mode.

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

38

Using Breakpoints

A breakpoint marks a place in the test script where you want to pause a test run. Breakpoints help to identify flaws in a script.

This chapter describes:

- About Using Breakpoints
- Choosing a Breakpoint Type
- Setting Break at Location Breakpoints
- Setting Break in Function Breakpoints
- Modifying Breakpoints
- Deleting Breakpoints

About Using Breakpoints

By setting a breakpoint you can stop a test run at a specific place in a test script. A breakpoint is indicated by a breakpoint marker in the left margin of the test window.

WinRunner pauses the test run when it reaches a breakpoint. You can examine the effects of the test run up to the breakpoint, view the current value of variables, make any necessary changes, and then continue running the test from the breakpoint. You use the **Run from Arrow** command to restart the test run from the breakpoint. Once restarted, WinRunner continues running the test until it encounters the next breakpoint or the test is completed.

Note: WinRunner only pauses when it is not in batch mode. When running tests in batch mode, WinRUnner ignores breakpoints.

Breakpoints are useful for:

- suspending the test run at a certain point and inspecting the state of your application.
- monitoring the entries in the Watch List. See Chapter 39, "Monitoring Variables," for more information.
- marking a point from which to begin stepping through a test script using the Step commands. See Chapter 37, "Controlling Your Test Run," for more information.

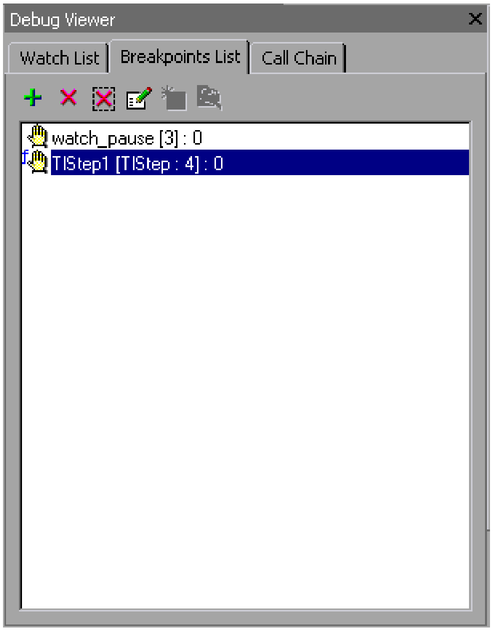
There are two types of breakpoints: Break at Location and Break in Function. A Break at Location breakpoint stops a test at a specified line number in a test script. A Break in Function breakpoint stops a test when it calls a specified user-defined function in a loaded compiled module.

You set a pass count for each breakpoint you define. The pass count determines the number of times the breakpoint is passed before it stops the test run. For example, suppose you program a loop that performs a command twenty-five times. By default, the pass count is set to zero, so test execution stops after each loop. If you set the pass count to 25, execution stops only after the twenty-fifth iteration of the loop.

Note: The breakpoints you define are active only during your current WinRunner session. If you terminate your WinRunner session, you must redefine breakpoints to continue debugging the script in another session.

Viewing the Breakpoints List in the Debug Viewer

You view the values of variables in the Breakpoints List pane in the Debug Viewer window. If the Debug Viewer window is not currently displayed, or the Breakpoints List pane is not open in the window, choose **Debug > Breakpoints List** to display it. If the Breakpoints List pane is open, but a different pane is currently displayed, click the **Breakpoints List** tab to display it.



watch_pause [3] : 0
TlStep1 [TlStep : 4] : 0

Tip: The Debug Viewer window can be displayed as a docked window within the WinRunner window, or it can be a floating window that you can drag to any location on your screen. By default the Debug Viewer opens as a docked window on the right side of the WinRunner screen. To move the window to another location, drag the Debug Viewer titlebar.

Choosing a Breakpoint Type

WinRunner enables you to set two types of breakpoints: Break at Location and Break in Function.

Break at Location

A Break at Location breakpoint stops a test at a specified line number in a test script. This type of breakpoint is defined by a test name and a test script line number. The breakpoint marker appears in the left margin of the test script, next to the specified line. A Break at Location breakpoint might, for example, appear in the Breakpoints List pane as:

```
ui_test[137] : 0
```

This means that the breakpoint marker appears in the test named *ui_test* at line 137. The number after the colon represents the pass count, which is set here to zero (the default). This means that WinRunner will stop running the test every time it passes the breakpoint.

Break in Function

A Break in Function breakpoint stops a test when it calls a specified user-defined function in a loaded compiled module. This type of breakpoint is defined by the name of a user-defined function and the name of the compiled module in which the function is located. When you define a Break in Function breakpoint, the breakpoint marker appears in the left margin of the WinRunner window, next to the first line of the function. WinRunner halts the test run each time the specified function is called. A Break in Function breakpoint might appear in the Breakpoints List pane as:

```
ui_func [ui_test : 25] : 10
```

This indicates that a breakpoint has been defined for the line containing the *ui_func* function, in the *ui_test* compiled module: in this case line 25. The pass count is set to 10, meaning that WinRunner stops the test each time the function has been called ten times.

Setting Break at Location Breakpoints

You set Break at Location breakpoints using the Breakpoints List pane in the Debug Viewer, the mouse, or the Toggle Breakpoint command.

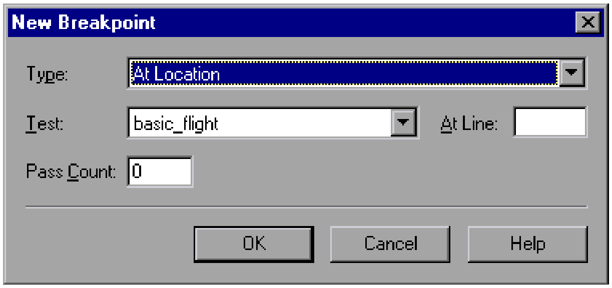
Note: You can set a breakpoint in a function only after the function has been loaded into WinRunner (the function has been executed at least once).

To set a Break at Location breakpoint using the Breakpoints List pane:

1 Display the Breakpoints List as described in “Viewing the Breakpoints List in the Debug Viewer” on page 729.

2 Click **Add Entry** to open the New Breakpoint dialog box.

3 In the **Type** box, select **At Location**.



New Breakpoint

Type: At Location

Test: basic_flight

At Line:

Pass Count: 0

4 The **Test** box displays the name of the active test. If you want to insert a breakpoint for another test, select the name from the **Test** list.

5 Enter the line number at which you want to add the breakpoint in the **At Line** box

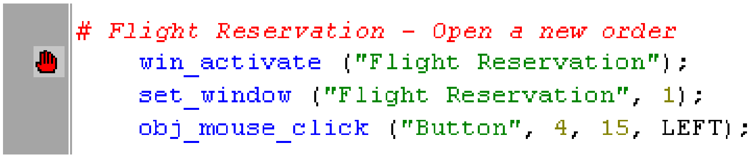
6 If you want the test to break each time it reaches the breakpoint, accept the default **Pass Count**, 0. If you only want the test to break after it reaches the breakpoint a given number of times, enter the number in the **Pass Count** box.

7 Click **OK** to set the breakpoint and close the New Breakpoint dialog box. The new breakpoint is displayed in the Breakpoints List pane.

The breakpoint marker appears in the left margin of the test script, next to the specified line.

To set a Break at Location breakpoint using the mouse:

1 Right-click the left (gray) margin of the WinRunner window next to the line where you want to add a breakpoint. The breakpoint symbol appears in the left margin of the WinRunner window:



```
# Flight Reservation - Open a new order
    win_activate ("Flight Reservation");
    set_window ("Flight Reservation", 1);
    obj_mouse_click ("Button", 4, 15, LEFT);
```

Tip: If the gray margin is not visible, choose **Tools** > **Editor Options** and click the **Options** tab. Then select the **Visible gutter** option.

---

2 Breakpoints added using this method automatically use a pass count of 0. If you want to use a different pass count, modify the breakpoint as described in "Modifying Breakpoints" on page 734.

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

1 Move the insertion point to the line of the test script where you want test execution to stop.

2 Choose **Debug** > **Toggle Breakpoint** or click the **Toggle Breakpoint** button. The breakpoint symbol appears in the left margin of the WinRunner window and is displayed in the Breakpoints List.

3 Breakpoints added using this method automatically use a pass count of 0. If you want to use a different pass count, modify the breakpoint as described in "Modifying Breakpoints" on page 734.

To remove a Break at Location breakpoint:

Right-click the breakpoint symbol

or:

Choose **Debug** > **Toggle Breakpoint**, or click the **Toggle Breakpoint** button.

Setting Break in Function Breakpoints

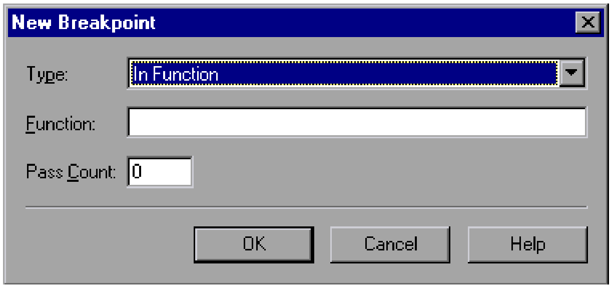
A Break in Function breakpoint stops test execution at the user-defined function that you specify. You set a Break in Function breakpoint from the Breakpoint Lists pane in the Debug Viewer, or the **Break in Function** command.

Note: You can set a breakpoint in a function only after the function has been loaded into WinRunner (the function has been executed at least once).

To set a Break in Function breakpoint:

1. If you want to set a break in function breakpoint for a function that is already a part of your test, place the insertion point on the function name.
2. Choose **Debug** > **Break in Function**. The New Breakpoint dialog box opens. Proceed to step 5.
3. Alternatively, you can open the New Breakpoint dialog box from the Breakpoint Lists pane. Display the Breakpoints List as described in “Viewing the Breakpoints List in the Debug Viewer” on page 729.
4. Click **Add Entry.**

5 The New Breakpoint dialog box opens.



New Breakpoint

Type: In Function

Function:

Pass Count: 0

Accept the breakpoint type: **In Function**.

6 By default, the **Function** box displays the name of the function (or text) in which the insertion point is currently located. Accept the function name or enter the name of a valid function. The function name you specify must be compiled by WinRunner. For more information, see Chapter 29, "Creating User-Defined Functions," and Chapter 30, "Creating Compiled Modules."

7 Type a value in the **Pass Count** box.

8 Click **OK** to set the breakpoint and close the New Breakpoint dialog box.

The new breakpoint is displayed in the Breakpoints List pane.

The breakpoint symbol is displayed in the left margin next to the first line of the function in the compiled module .

Modifying Breakpoints

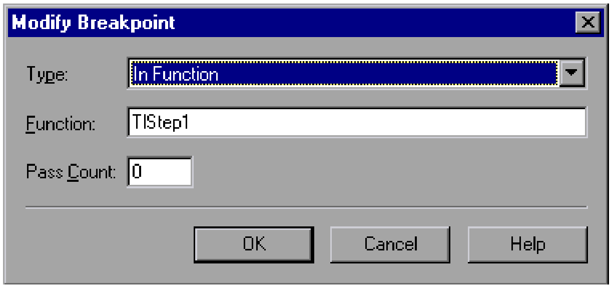
You can modify the definition of a breakpoint using the Modify Breakpoints dialog box. You can change the breakpoint's type, the test or line number for which it is defined, and the value of the pass count.

To modify a breakpoint:

1 Display the Breakpoints List as described in "Viewing the Breakpoints List in the Debug Viewer" on page 729.

2 Select a breakpoint in Breakpoint Lists pane.

3 Click **Modify entry** to open the Modify Breakpoint dialog box.



Modify Breakpoint

Type: In Function

Function: TIStep1

Pass Count: 0

4 To change the type of breakpoint, select a different breakpoint type in the **Type** box.

5 Change the settings as necessary.

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

Deleting Breakpoints

You can delete a single breakpoint or all breakpoints defined for the current test using the Breakpoints dialog box.

To delete a single breakpoint:

1 Display the Breakpoints List as described in “Viewing the Breakpoints List in the Debug Viewer” on page 729.

2 Select a breakpoint from the list.

3 Click **Delete entry**. The breakpoint is removed from the list and the breakpoint symbol is removed from the left margin of the test.

To delete all breakpoints using the Delete All Breakpoints Command:

Choose **Debug** > **Delete All Breakpoints** or click the **Delete All Breakpoints** toolbar button.

To delete all breakpoints using the Debug Viewer:

1. Display the Breakpoints List as described in "Viewing the Breakpoints List in the Debug Viewer" on page 729.
2. Click **Delete all breakpoints**. All breakpoints are deleted from the list and all breakpoint symbols are removed from the left margin of the relevant tests.

39

Monitoring Variables

The Watch List displays the values of variables, expressions, and array elements during a test run. You use the Watch List to enhance the debugging process.

This chapter describes:

➤ About Monitoring Variables

➤ Adding Variables to the Watch List

➤ Viewing Variables in the Watch List

➤ Modifying Variables in the Watch List

➤ Assigning a Value to a Variable in the Watch List

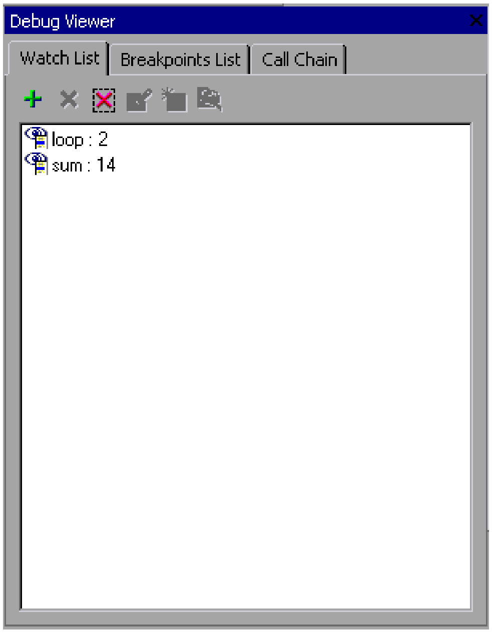
➤ Deleting Variables from the Watch List

About Monitoring Variables

The Watch List enables you to monitor the values of variables, expressions, and array elements while you debug a test script. Prior to running a test, you add the elements that you want to monitor to the Watch List. At each break during a test run—such as after a Step command, at a breakpoint, or at the end of a test, you can view the current values of the entries in the Watch List.

Viewing the Watch List in the Debug Viewer

You view the values of variables in the Watch List pane in the Debug Viewer window. If the Debug Viewer window is not currently displayed, or the Watch List pane is not open in the window, choose **Debug** > **Watch List** to display it. If the Watch List pane is open, but a different pane is currently displayed, click the **Watch List** tab to display it.

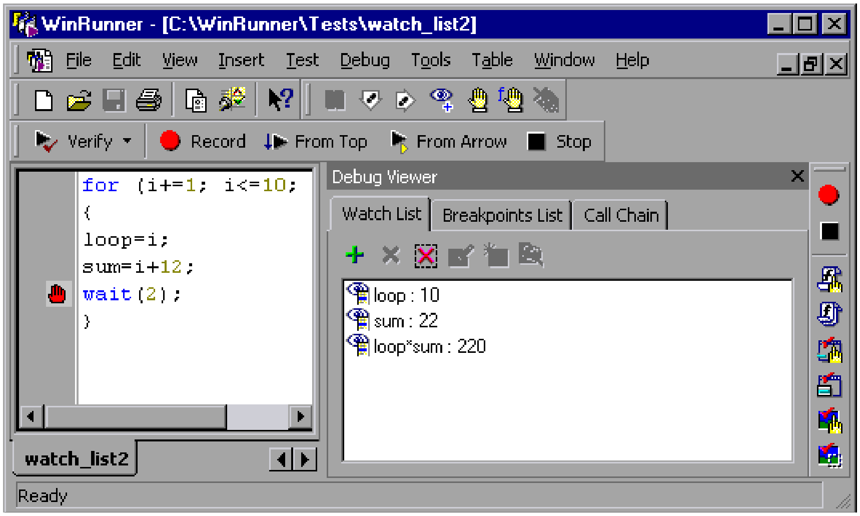


loop : 2
sum : 14

Tip: The Debug Viewer window can be displayed as a docked window within the WinRunner window, or it can be a floating window that you can drag to any location on your screen. By default the Debug Viewer opens as a docked window on the right side of the WinRunner screen. To move the window to another location, drag the Debug Viewer titlebar.

Watching Variable Values—An Example

For example, in the following test, the Watch List is used to measure and track the values of variables *loop* (the current loop*)* and *sum*. On the last step of each loop, the test pauses at the breakpoint so you can view the current values.



```
for (i+=1; i<=10;
{
loop=i;
sum=i+12;
wait(2);
}
```

```
loop : 10
sum : 22
loop*sum : 220
```

After WinRunner executes the first loop, the test pauses. The Watch List displays the variables and updates their values: When WinRunner completes the test run, the Watch List shows the following results:

```
loop:10
sum:22
loop*sum:220
```

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

If you choose a test or function in the Call Chain list (**Debug** > **Call Chain**), the context of the variables and expressions in the Watch List changes. WinRunner automatically updates their values in the Watch List.

Adding Variables to the Watch List

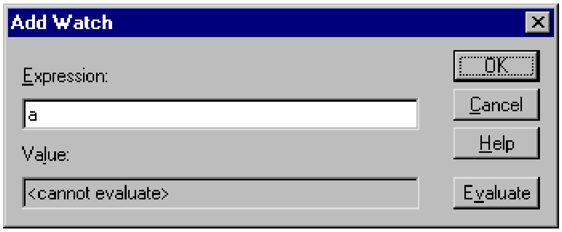
You add variables, expressions, and arrays to the Watch List using the Add Watch dialog box. You can add entries before running a test or when the test breaks after a Step command, when the test is paused, or at a breakpoint.

To add a variable, an expression, or an array to the Watch List:

1 Choose **Debug** > **Add Watch** or click the **Add Watch** button.

Alternatively, display the Watch List as described in "Viewing the Watch List in the Debug Viewer" on page 738 and click **Add entry**.

2 The Add Watch dialog box opens.



Add Watch

Expression: a

Value: <cannot evaluate>

In the **Expression** box, enter the variable, expression, or array that you want to add to the Watch List.

3 Click **Evaluate** to see the current value of the new entry. If the new entry contains a variable or an array that has not yet been initialized, the message "<cannot evaluate>" appears in the **Value** box. The same message appears if you enter an expression that contains an error.

4 Click **OK**. The Add Watch dialog box closes and the new entry appears in the **Watch List**.

Note: Do not add expressions that assign or increment the value of variables to the Watch List; this can affect the test run.

Viewing Variables in the Watch List

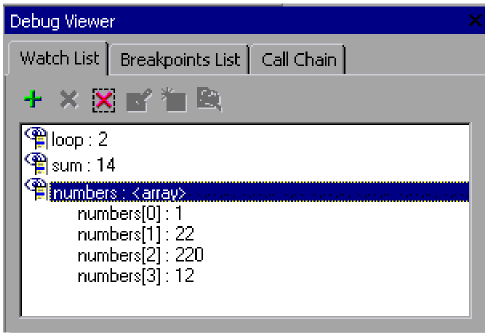
Once you add variables, expressions, and arrays to the Watch List, you can use the Watch List to view their values.

To view the values of variables, expressions, and arrays in the Watch List:

1 Display the Watch List as described in “Viewing the Watch List in the Debug Viewer” on page 738.

The variables, expressions and arrays are displayed; current values appear after the colon.

2 To view values of array elements, double-click the array name. The elements and their values appear under the array name. Double-click the array name to hide the elements.



```
loop : 2
sum : 14
numbers : <array>
    numbers[0] : 1
    numbers[1] : 22
    numbers[2] : 220
    numbers[3] : 12
```

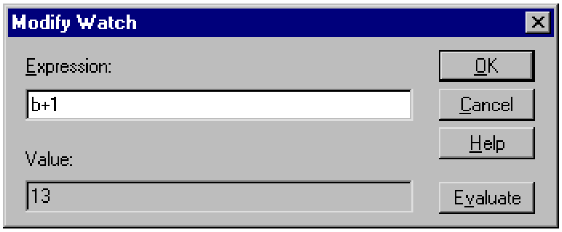
3 Click **Close**.

Modifying Variables in the Watch List

You can modify variables and expressions in the Watch List using the Modify Watch dialog box. For example, you can turn variable *b* into the expression *b + 1*, or you can change the expression *b + 1* into *b * 10*. When you close the Modify Watch dialog box, the Watch List is automatically updated to reflect the new value for the expression.

To modify an expression in the Watch List:

1. Display the Watch List as described in "Viewing the Watch List in the Debug Viewer" on page 738.
2. Select the variable or expression you want to modify.
3. Click **Modify entry** to open the Modify Watch dialog box.



Expression: b+1

Value: 13

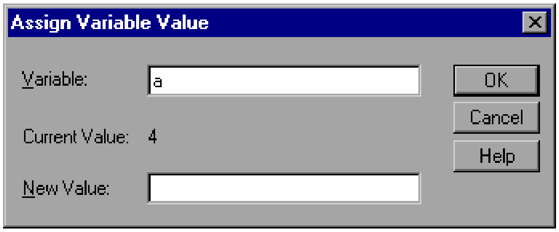
4. Change the expression in the **Expression** box as needed.
5. Click **Evaluate**. The new value of the expression appears in the **Value** box.
6. Click **OK** to close the Modify Watch dialog box. The modified expression and its new value appear in the Watch List.

Assigning a Value to a Variable in the Watch List

You can assign new values to variables and array elements in the Watch List. Values can be assigned only to variables and array elements, not to expressions.

To assign a value to a variable or an array element:

1. Display the Watch List as described in “Viewing the Watch List in the Debug Viewer” on page 738.
2. Select a variable or an array element.
3. Click **Assign Variable Value** to open the Assign Variable Value dialog box.



Assign Variable Value

Variable: a

Current Value: 4

New Value:

4. Type the new value for the variable or array element in the **New Value** box.
5. Click **OK** to close the dialog box. The new value appears in the Watch List.

Deleting Variables from the Watch List

You can delete selected variables, expressions, and arrays from the Watch List, or you can delete all the entries in the Watch List.

To delete a variable, an expression, or an array:

1. Display the Watch List as described in “Viewing the Watch List in the Debug Viewer” on page 738.
2. Select a variable, an expression, or an array to delete.

Note: You can delete an array only if its elements are hidden. To hide the elements of an array, double-click the array name in the Watch List.

3 Click **Delete entry** to remove the entry from the list.

4 Click **Close** to close the Watch List dialog box.

To delete all entries in the Watch List:

1 Display the Watch List as described in "Viewing the Watch List in the Debug Viewer" on page 738.

2 Click **Delete all entries**. All entries are deleted.

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

Part VII

Configuring WinRunner

40

Setting Properties for a Single Test

The Test Properties dialog box enables you to set properties for a single test. You set test properties to store information about a WinRunner test and to control how WinRunner runs a test.

This chapter describes:

- About Setting Properties for a Single Test
- Setting Test Properties from the Test Properties Dialog Box
- Documenting General Test Information
- Documenting Descriptive Test Information
- Managing Test Parameters
- Associating Add-ins with a Test
- Reviewing Current Test Settings
- Defining Startup Applications and Functions

About Setting Properties for a Single Test

You can set test properties to document information about a specific test, or that specify your preferences for a specific test. For example, you can enter a detailed description of the test, or indicate the add-ins required for a test.

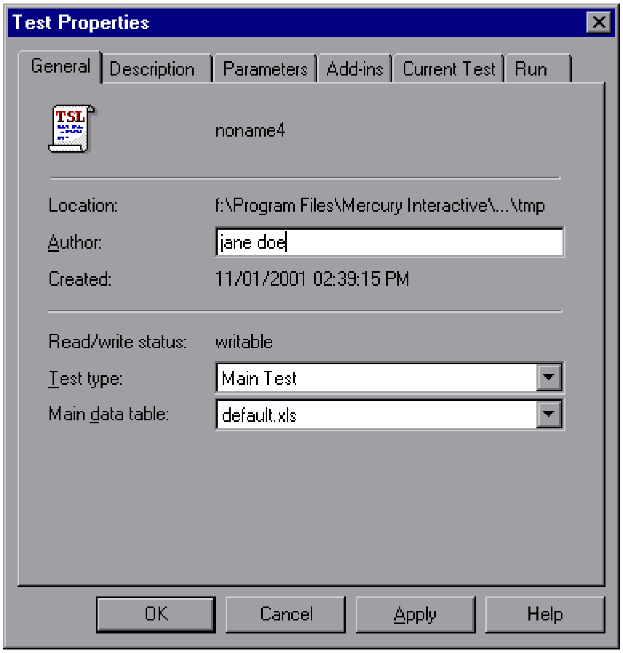
Setting Test Properties from the Test Properties Dialog Box

When you create a test, you can set test properties for the test.

To set test properties:

1 Choose **File** > **Test Properties**.

The Test Properties dialog box opens. It is divided by subject into six tabbed pages.



Test Properties

Location: f:\Program Files\Mercury Interactive\...\tmp

Author: jane doe

Created: 11/01/2001 02:39:15 PM

Read/write status: writable

Test type: Main Test

Main data table: default.xls

2 To set the properties for your test, select the appropriate tab and set the options, as described in the sections that follow.

3 To apply your changes and keep the Test Properties dialog box open, click **Apply**.

4 When you are finished, click **OK** to apply your changes and close the dialog box.

The Test Properties dialog box contains the following tabbed pages:

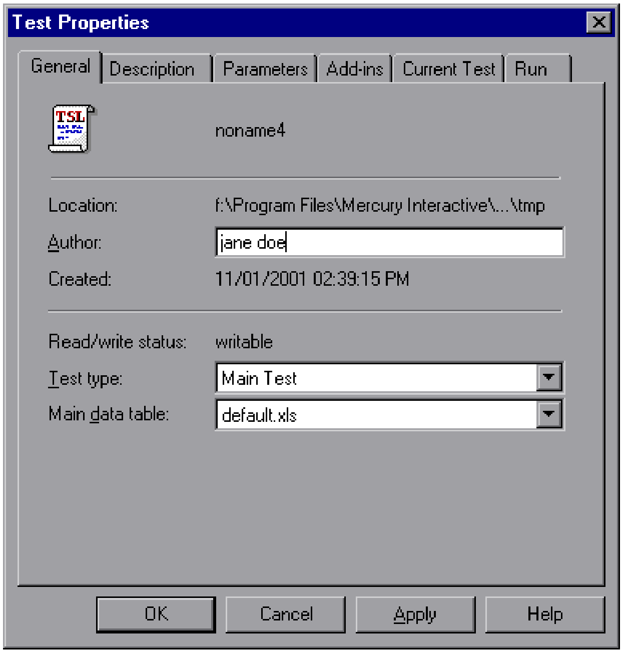
Tab Heading	Description
General	Enables you to set general information about the test.
Description	Enables you to enter descriptive information about the test.
Parameters	Enables you to define test parameters.
Add-ins	Enables you to indicate the add-ins required for the test.
Current Test	Enables you to review the current folder and run mode settings for the test.
Run	Enables you to define startup applications and functions.

You can also set testing options that affect all tests. For more information, see Chapter 41, “Setting Global Testing Options.”

Documenting General Test Information

You can document and view general information about a test in the **General** tab of the Test Properties dialog box. For example, you can enter the name of the test author and choose whether the test is a main test or a compiled module.

When you are connected to TestDirector and have a test open from a TestDirector project, the test's system file path and version control information are also shown.



noname4

Location: f:\Program Files\Mercury Interactive\...\tmp

Author: jane doe

Created: 11/01/2001 02:39:15 PM

Read/write status: writable

Test type: Main Test

Main data table: default.xls

This tab contains the following information:

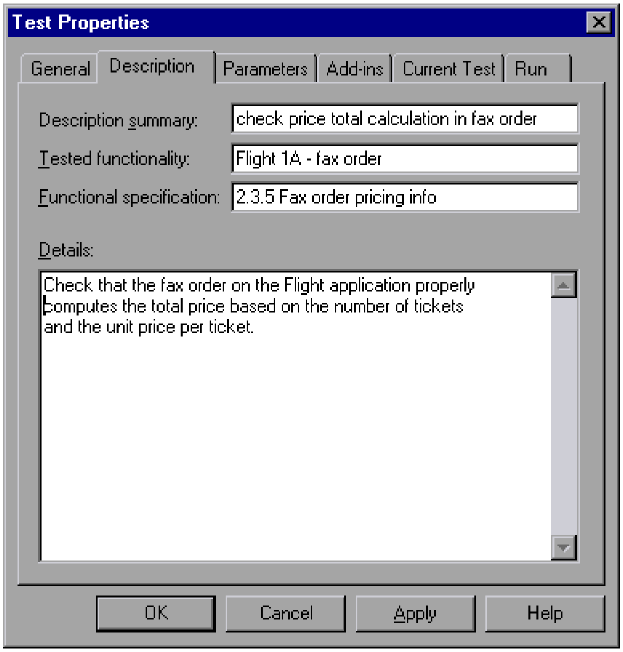
Option	**Description**
TSL	Displays the name of the test.
Location	Displays the test's location within the TestDirector tree if the test is stored in TestDirector. Otherwise, this field displays the test's location within the file system.
Author	Enables you to specify the test author's name.
Created	Displays the date and time that the test was created.

Option	Description
Read/write status	Indicates whether the test is read-only (either the test folder or the script is marked as read-only in the file system) or writable. If the test is read-only, all editable property fields in the Test Properties dialog box are disabled.
Test type	Indicates whether the test is a Main (standard) Test or a Compiled Module. For more information about compiled modules, see "Creating a Compiled Module," on page 594.
Main data table	Contains the main data table for the test. For more information, see "Assigning the Main Data Table for a Test," on page 453.
File system path	Displays the system file path of the test. This information is only displayed when you are connected to TestDirector and the current test is opened from a TestDirector project.
Version control	Displays version control information for the test. This information is only displayed when you are connected to a version of TestDirector that supports version control and the current test is opened from a TestDirector project.

Documenting Descriptive Test Information

You can document descriptive information about the test in the **Description** tab of the Test Properties dialog box. You can enter a summary description of the test, the application feature(s) you are testing, and a reference to the relevant functional specifications document(s).

You can also enter a detailed description of the test.



Description summary: check price total calculation in fax order

Tested functionality: Flight 1A - fax order

Functional specification: 2.3.5 Fax order pricing info

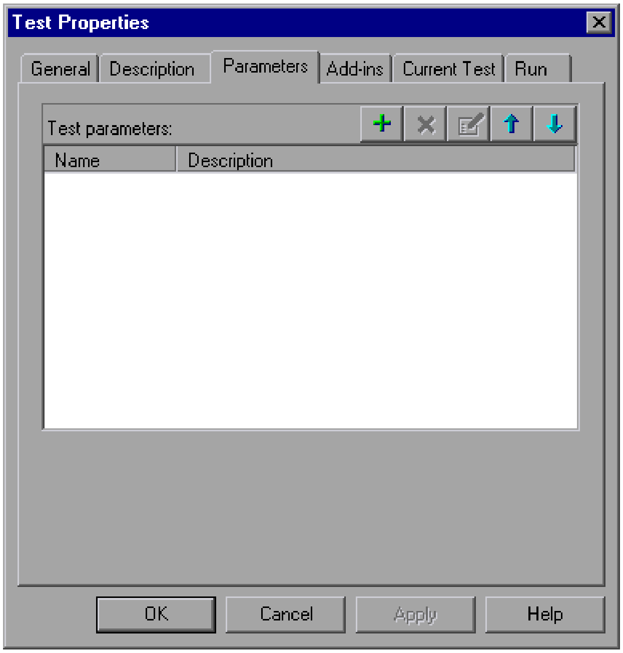
Details:
Check that the fax order on the Flight application properly
computes the total price based on the number of tickets
and the unit price per ticket.

This tab contains the following information:

Option	Description
Description summary	Enables you to specify a short summary of the test.
Tested functionality	Enables you to specify a description of the application functionality you are testing.
Functional specification	Enables you to specify a reference to the application’s functional specification(s).
Details	Enables you to specify a detailed description of the test.

Managing Test Parameters

You can manage test parameters by adding (declaring), modifying, and deleting parameters in the **Parameters** tab of the Test Properties dialog box.



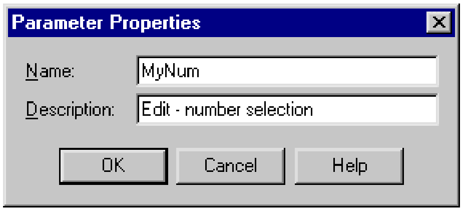
The Test Parameters list displays the existing test parameters. When your test is called by another test, the parameters that are listed in the Parameters tab are assigned the values supplied by the calling test.

Note that you must declare your test parameters in this dialog box in order to receive parameter values from a calling test.

For more information about parameters, see “Replacing Data from the Test with Parameters” on page 575.

To define a new parameter:

1 In the **Parameters** tab of the Test Properties dialog box, click the **Add** button. The Parameter Properties dialog box opens.



Name: MyNum

Description: Edit - number selection

2 Enter a **Name** and a **Description** for the parameter.

3 Click **OK**. The parameter is added to the **Test parameters** list.

4 Use the **Up** and **Down** arrow buttons to change the order of the parameters.

Note: Because parameter values are assigned sequentially, the order in which parameters are listed in the Parameters tab determines the value that is assigned to a parameter by the calling test.

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

To delete a parameter from the parameter list:

1 In the **Parameters** tab of the Test Properties dialog box, select the name of the parameter to delete.

2 Click the **Delete** button.

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

To modify a parameter in the parameter list:

1 In the **Parameters** tab of the Test Properties dialog box, select the name of the parameter to modify.

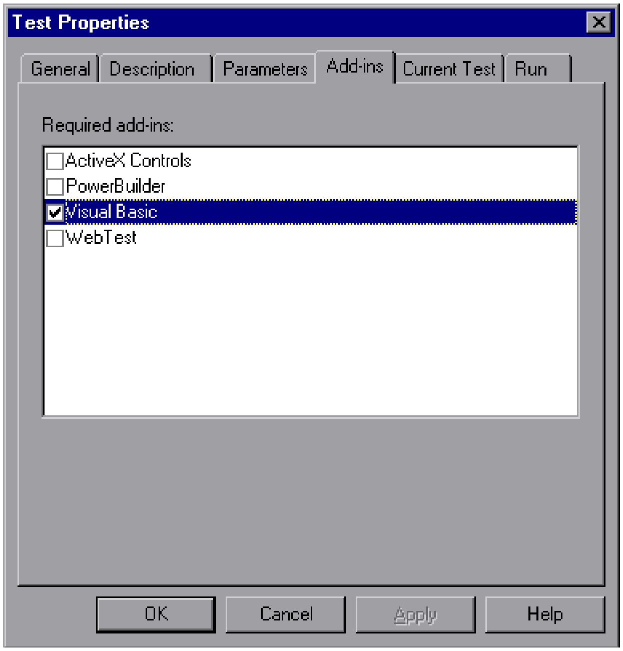
2 Click the **Modify** button. The Parameter Properties dialog box opens with the current name and description of the parameter.

3 Modify the parameter as needed.

4 Click **OK** to close the dialog box. The modified parameter is displayed in the **Test parameters** list.

Associating Add-ins with a Test

You can indicate the WinRunner add-ins that are required for a test by selecting them in the **Add-ins** tab of the Test Properties dialog box.



Required add-ins:

□ ActiveX Controls
□ PowerBuilder
☑ Visual Basic
□ WebTest

The **Add-ins** tab contains one check box for each add-in you currently have installed. When you begin creating a new test, the add-ins that are loaded at that time are automatically selected as the required add-ins. You can indicate which add-ins the test actually requires by changing the selected check boxes. This information reminds you or others which add-ins to load in order to successfully run this test. It also instructs TestDirector to confirm that the selected add-ins are loaded. For more information, see “Running Tests with Add-ins from TestDirector” on page 756.

Note: You can see which add-ins are loaded at any time in the **About WinRunner** dialog box. Loaded add-ins are marked with a "+".

To associate add-ins with a test:

1. Choose **File** > **Test Properties** to open the Test Properties dialog box.
2. Click the **Add-ins** tab.
3. Select the add-in(s) that are required for the test.

Running Tests with Add-ins from TestDirector

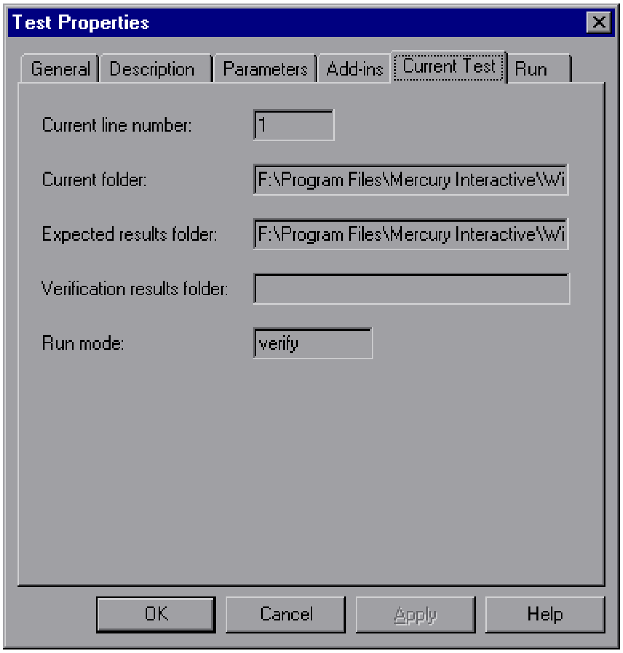
In addition to providing information for people running your test from WinRunner, the **Add-ins** tab instructs TestDirector to load the selected add-ins when it runs WinRunner tests.

When you run a test from TestDirector, TestDirector will load the add-ins selected in the **Add-ins** tab for the test. If WinRunner is already open, but the required add-ins are not loaded, TestDirector closes and reopens WinRunner with the proper add-ins. If one or more of the required add-ins are not installed, TestDirector displays a "Cannot open test." error message.

For more information about running WinRunner tests from TestDirector, refer to the *TestDirector User's Guide*.

Reviewing Current Test Settings

You can review the folder and run mode information for the current test in a read-only view in the **Current Test** tab of the Test Properties dialog box.



Current line number: 1

Current folder: F:\Program Files\Mercury Interactive\Wi

Expected results folder: F:\Program Files\Mercury Interactive\Wi

Verification results folder:

Run mode: verify

Current line number

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 44, “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 44, “Setting Testing Options from a Test Script.”

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 44, "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 36, "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 44, "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 44, "Setting Testing Options from a Test Script."

Defining Startup Applications and Functions

Startup applications and functions are applications and functions that WinRunner runs and executes before running a test. For example, you can set the Flight Reservation application as your startup application and you can define a startup function that logs in to the Flight Reservation application before your test run begins.

You define startup applications and startup functions in the **Run** tab of the Test Properties dialog box. You can define startup application and/or function options while creating your test. You can also select whether or not to run the startup application and/or function before running your test without modifying the startup function or application definitions.

WinRunner implements **Run** tab settings only when you run the test from the beginning, such as when you select **Run From Top** or **Run Minimized** > **From Top**, or when the test runs . For more information on these options, see "WinRunner Run Commands" on page 625.

WinRunner implements the **Run** tab settings of a called test when the called test runs, unless you use **Step Into** to open the called test. For more information on calling a test, see Chapter 28, "Calling Tests." For more information on the **Step Into** option, see Chapter 37, "Controlling Your Test Run."

Note: If you choose to run an application and execute a function before the test begins, the startup application runs before the startup function executes.

Defining a Startup Application

When defining a startup application, you specify the path to the application, any required parameters, and the amount of time WinRunner waits between invoking the application and running the test.

Note that additional methods exist for running an application:

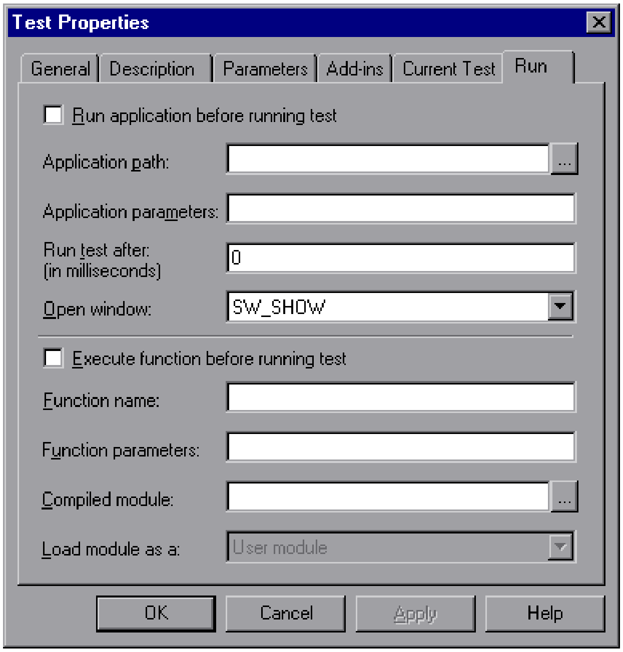
- You can use the **invoke_application** function to run an application at any time from within a test script. Use this method to run an application during a test run. For more information, see "Starting Applications from a Test Script" on page 552.
- You can run an application when you run WinRunner from the command line. Use this method to run the application before WinRunner starts. For more information, see Chapter 36, "Running Tests from the Command Line."

Note: If the application specified as the startup application is already running when you run your test, WinRunner will not open a new instance of the application at the beginning of the test.

To define a startup application:

1 Choose **File** > **Test Properties** to open the Test Properties dialog box.

2 Click the **Run** tab.



Test Properties

General | Description | Parameters | Add-ins | Current Test | Run

□ Run application before running test

Application path: [ ] ...

Application parameters: [ ]

Run test after: (in milliseconds) 0

Open window: SW_SHOW

□ Execute function before running test

Function name: [ ]

Function parameters: [ ]

Compiled module: [ ] ...

Load module as a: User module

OK | Cancel | Apply | Help

3 Select the **Run application before running test** check box if you want your startup application to run in the next test run.

...

4 In the **Application path** box, enter the application path or use the browse button to navigate to the application that you want to run.

5 Enter any required application parameters in the **Application parameters** box, separated by commas (,). For information about application parameters, refer to the application documentation.

6 In the **Run test after** box, enter the amount of time you want the system to wait between invoking the application and running the test, or accept the default (0 milliseconds).

7 In the **Open window** box, select how you want the application window to appear when it opens. The possible options are:

Option	**Description**
SW_HIDE	Hides the window and activates another window.
SW_SHOWNORMAL	Activates and displays a window. If the window is minimized or maximized, Windows restores it to its original size and position. Specify this flag when displaying the window for the first time.
SW_SHOWMINIMIZED	Activates the window and displays it as a minimized window.
SW_SHOWMAXIMIZED	Activates the window and displays it as a maximized window.
SW_SHOWNOACTIVATE	Displays a window in its most recent size and position. The active window remains active.
SW_SHOW	Activates the window and displays it in its current size and position.
SW_MINIMIZE	Maximizes the specified window and activates the next top-level window in the z-order.
SW_SHOWMINNOACTIVE	Displays the window as a minimized window. The active window remains active.
SW_SHOWNA	Displays the window in its current state. The active window remains active.
SW_RESTORE	Activates and displays the window. If the window is minimized or maximized, Windows restores it to its original size and position. Specify this flag when restoring a minimized window.

Note: You can also set this option using the **-app_open_win** command line option. For more information, see Chapter 36, "Running Tests from the Command Line."

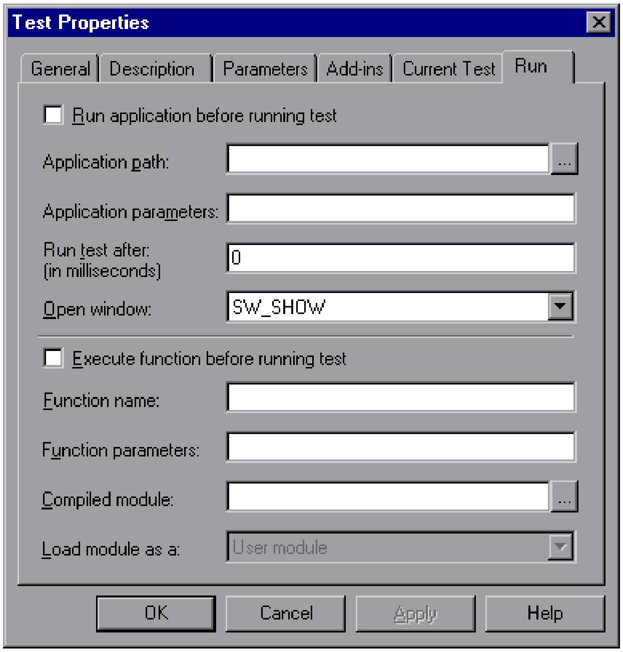
Defining a Startup Function

A startup function can be either a TSL function or a user-defined function contained in a compiled module. When defining a startup function, you specify the name of the function, function parameters (if any), and the compiled module name and type (for user-defined functions). For more information on TSL functions, see Chapter 27, "Generating Functions" and refer to the *TSL Reference*. For more information on user-defined functions and compiled modules, see Chapter 29, "Creating User-Defined Functions" and Chapter 30, "Creating Compiled Modules."

To define a startup function:

1 Choose **File** > **Test Properties** to open the Test Properties dialog box.

2 Click the **Run** tab.



Test Properties

General | Description | Parameters | Add-ins | Current Test | Run

□ Run application before running test

Application path:

Application parameters:

Run test after: (in milliseconds) 0

Open window: SW_SHOW

□ Execute function before running test

Function name:

Function parameters:

Compiled module:

Load module as a: User module

3 Select the **Execute function before running test** check box if you want your startup function to execute the next time your test runs.

4 In the **Function name** box, enter the name of the function.

Note: The function name can contain only alphanumeric characters and underscores and cannot begin with a number.

---

5 Enter any parameters required for the function in the **Function parameters** box.

6 If the function is part of a compiled module, enter the name of the compiled module containing the function in the **Compiled module** box, or use the browse button to navigate to the compiled module.

7 If the function is part of a compiled module, select the compiled module type in the **Load module as a** box. For more information on system and user modules, see "Loading and Unloading a Compiled Module" on page 595.

41

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:

About Setting Global Testing Options

WinRunner testing options affect how you record test scripts and run tests. The options also affect the way WinRunner opens and the way the main window appears. For example, you can set the speed at which WinRunner runs a test, determine how WinRunner records keyboard input, or select a background style for the WinRunner main window.

You set these and other options for all tests using the General Options dialog box.

You can also set and retrieve some options during a test run 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 44, "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.

The General Options dialog box is composed of an *options tree* and an *options pane*. Clicking a category or sub-category in the options tree displays the corresponding options in the options pane.

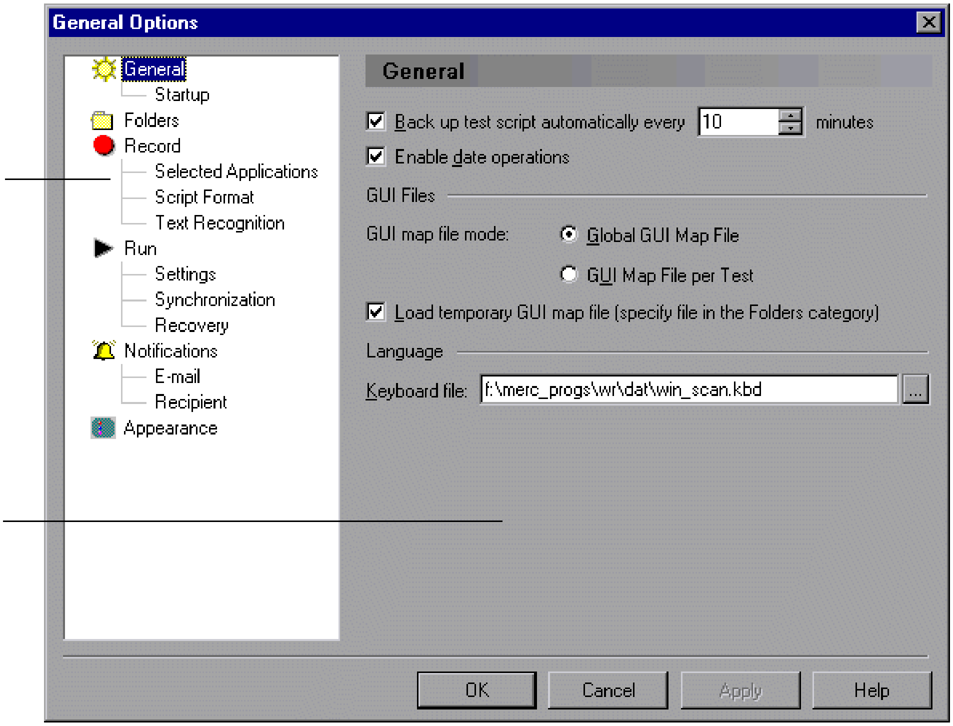
The General Options dialog box contains the following categories and sub-categories:

Category	Subject
General	Contains options for GUI map preferences, language settings and other general testing options.
Startup	Contains options that control what happens when WinRunner opens.
Folders	Specifies the folder location of WinRunner files and the search paths for resolving relative paths.
Record	Contains options for recording tests.
Selected Applications	Contains options for choosing which applications you want to record.

Category	Subject
Script Format	Contains options for controlling the appearance and readability of your script.
Text Recognition	Contains options for recognizing text in your application.
Run	Contains options for running your test.
Settings	Contains settings for handling specific situations during the test run.
Synchronization	Defines synchronization settings for your test run.
Recovery	Contains options for specifying recovery and Web exception files.
Notifications	Enables you to specify the criteria for sending e-mail notifications.
E-mail	Contains options for specifying the mail server to use and other e-mail preferences.
Recipients	Enables you to specify the recipients to receive e-mail notifications.
Appearance	Contains options for controlling the appearance of WinRunner.

To set global testing options:

1. Choose **Tools** > **General Options**. The General Options dialog box opens.



Options Tree

Options Pane

General

☑ Back up test script automatically every 10 minutes

☑ Enable date operations

GUI Files

GUI map file mode: ● Global GUI Map File ○ GUI Map File per Test

☑ Load temporary GUI map file (specify file in the Folders category)

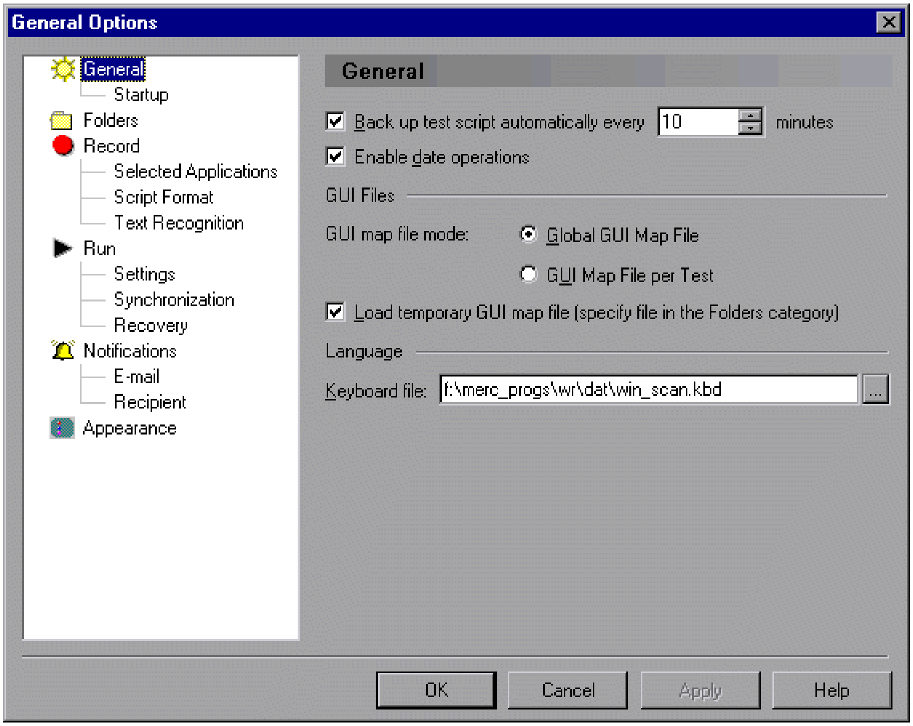
Language

Keyboard file: f:\merc_progs\wr\dat\win_scan.kbd

2. Click a category or subcategory in the options tree to display the corresponding options in the options pane.
3. Set the options you need, as described in the sections below.
4. To apply your changes and keep the General Options dialog box open, click **Apply**.
5. When you are finished, click **OK** to save your changes and close the dialog box.

Setting General Options

The **General** category contains options for GUI map preferences, language settings, and other general testing options.



General Options

General

☑ Back up test script automatically every 10 minutes

☑ Enable date operations

GUI Files

GUI map file mode: ● Global GUI Map File ○ GUI Map File per Test

☑ Load temporary GUI map file (specify file in the Folders category)

Language

Keyboard file: f:\merc_progs\wr\dat\win_scan.kbd

In addition to the options in this category, you can set additional recording options in the **Startup** subcategory.

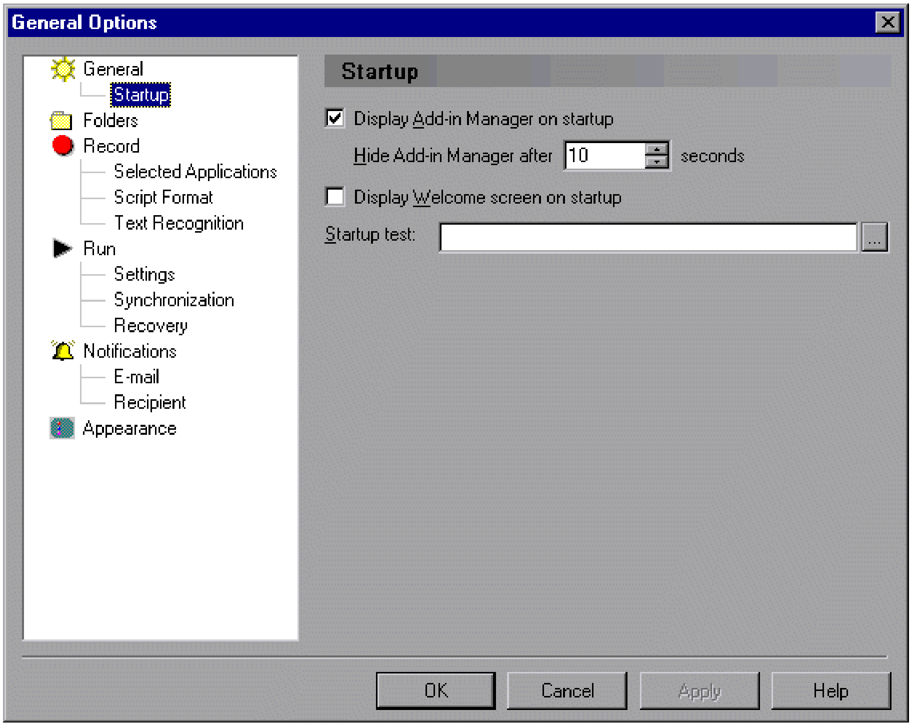
The **General** category contains the following options:

Option	Description
back up test script automatically every __ minutes	Instructs WinRunner to create a backup file for your script periodically, according to the specified interval. When selected, WinRunner creates a backup file in your test folder called *script.sav*, which is a simple text file of the script. Each time WinRunner backs up your script, it overwrites the previous *script.sav* file. **Default** = Selected **Default** = 10 [minutes]
Enable date operations	Enables date operation functionality and displays the **Tools** > **Date** menu item. **Note:** You must restart WinRunner for a change in this setting to take effect. **Default** = Cleared
GUI map file mode	Sets the GUI map file mode in WinRunner. • **Global GUI Map File**—enables you to create a GUI map file for your entire application, or for each window in your application. Multiple tests can reference a common GUI map file. For additional information, see Chapter 5, "Working in the Global GUI Map File Mode." • **GUI Map File per Test**— enables WinRunner to automatically create a GUI map file for each test you create. You do not need to worry about creating, saving, and loading GUI map files. For additional information, see Chapter 6, "Working in the GUI Map File per Test Mode." **Note:** You must restart WinRunner for a change in this setting to take effect. If you are working with tests created in WinRunner 6.02 or earlier, you must work in the *Global GUI Map File* mode. **Default** = Global GUI Map File

Option	Description
Load temporary GUI map file	Automatically loads the temporary GUI map file when starting WinRunner. **Note:** This option is disabled when the **GUI Map file per Test** option is selected, as there are no temporary GUI map files when working with separate GUI map files for each test. You can set the location of the temporary GUI map file in the **Folders** category of the General Options dialog box. **Default** = selected
Keyboard file	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** = **<WinRunner installation folder>\dat\win_scan.kbd**
Interface language	If WinRunner is installed on a non-English operating system, the Interface language option may be displayed. This option enables you to select the WinRunner interface language.

Setting Startup Options

The **Startup** category contains options that control what happens when WinRunner opens.



☑ Display Add-in Manager on startup

Hide Add-in Manager after 10 seconds

□ Display Welcome screen on startup

Startup test:

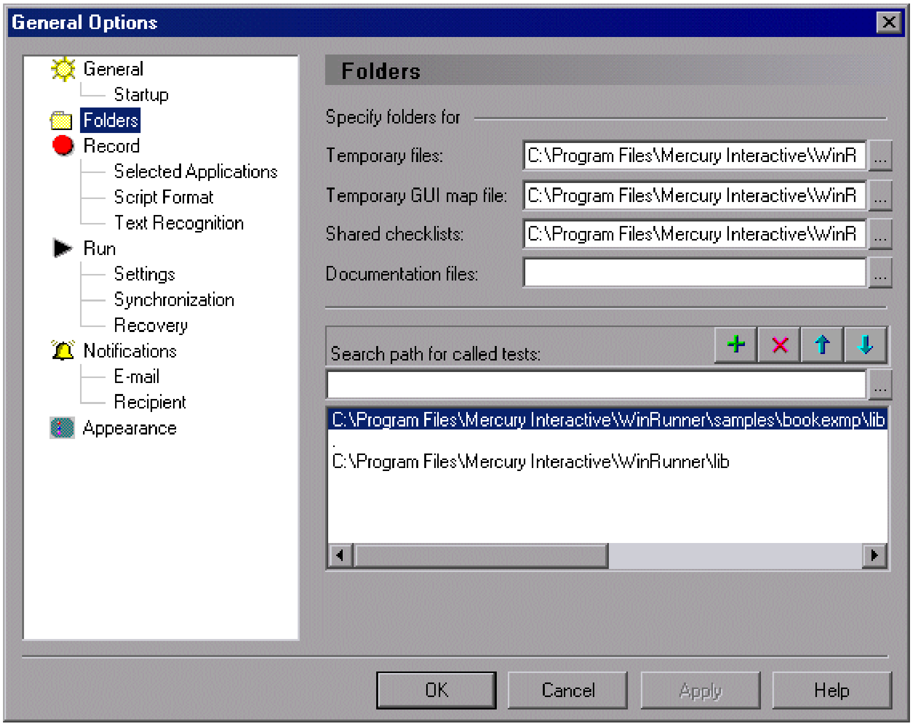
The **Startup** category contains the following options:

Option	Description
Display Add-in Manager on startup	Displays the Add-In Manager dialog box when starting WinRunner. For information about the Add-In Manager dialog box and loading installed add-ins when starting WinRunner, see “Loading WinRunner Add-Ins” on page 20. **Default** = Selected
Hide Add-in Manager after ___ seconds	Specifies how many seconds the Add-in Manager remains open before it closes and automatically loads the same add-ins that were loaded in the previous WinRunner session. **Default** = 10 seconds

Option	Description
Display Welcome screen on startup	Displays the Welcome screen when starting WinRunner. **Note:** Clearing the **Show on Startup** check box at the bottom of the Welcome screen also clears the selection in the **Startup** pane. **Default** = Selected
Startup test	Specifies the location of your startup test. You can use a startup test to perform operations such as configuring recording, loading compiled modules, and loading GUI map files when starting WinRunner. For more information, see Chapter 46, "Initializing Special Configurations." **Note:** You can also set the location of your startup test from the RapidTest Script wizard. A startup test can be used in addition to (and not instead of) the initialization (*tslinit*) test. You can specify a TestDirector script as your startup test. If you do, ensure that **Reconnect on startup** is selected in the TestDirector Connection dialog box. For more information, see "Connecting to and Disconnecting from a Project" on page 917. **Default** = **<WinRunner installation folder>**

Setting Folder Options

The **Folders** category enables you to specify the locations of WinRunner files and to specify search paths for resolving relative paths.



General Options

Folders

Specify folders for

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

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

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

Documentation files:

Search path for called tests:

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

.

C:\Program Files\Mercury Interactive\WinRunner\lib

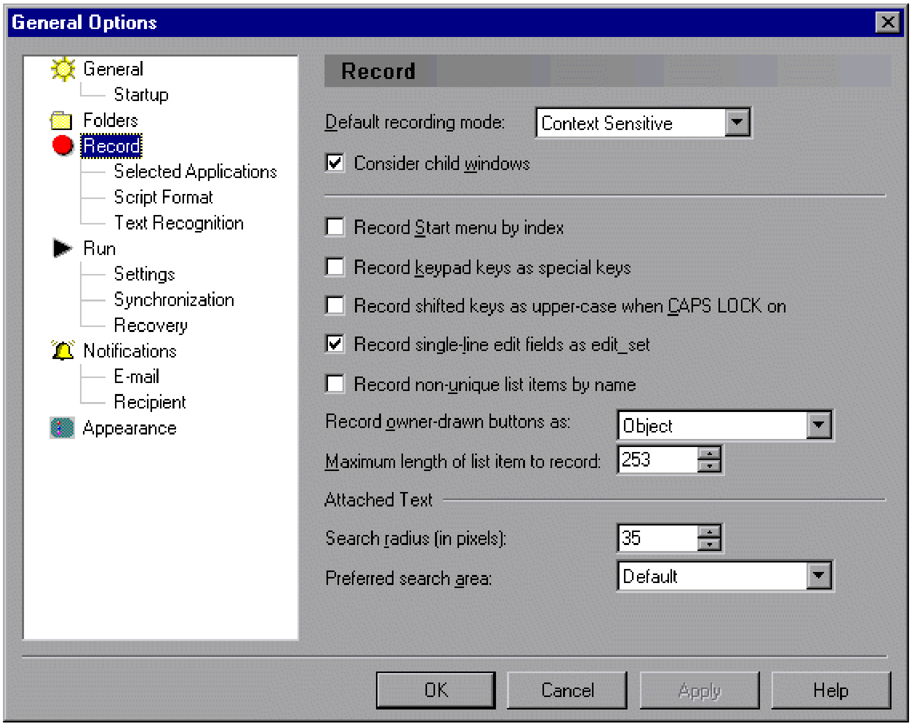
The **Folders** category contains the following options:

Option	Description
Temporary files	The folder containing temporary tests. Enter or browse to the folder. **Notes:** If you designate a new folder, you must restart WinRunner in order for the change to take effect. 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 44, "Setting Testing Options from a Test Script." **Default** = <*WinRunner installation folder*>\tmp
Temporary GUI map file	The folder containing the temporary GUI map file (*temp.gui*). If you select the **Load Temporary GUI Map File** check box in the **General** category 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:** If you designate a new folder, you must restart WinRunner in order for the change to take effect. Default = <*WinRunner installation folder*>\tmp
Shared checklists	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**, or **db_check** statement. To enter a new path, type it in the text box or click **Browse** to locate the folder. For more information on shared GUI checklists, see "Saving a GUI Checklist in a Shared Folder" on page 190. For more information on shared database checklists, see "Saving a Database Checklist in a Shared Folder" on page 350. **Notes:** If you designate a new folder, you must restart WinRunner in order for the change to take effect. 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 44, "Setting Testing Options from a Test Script." **Default** = <*WinRunner installation folder*>\chklist

Option	Description
Documentation files	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** = <*WinRunner installation folder*>\doc
Search path for called tests	The paths that WinRunner searches for called tests. If you define search paths in this pane, you can specify relative paths when calling tests. The order of the search paths in the list determines the order of locations in which WinRunner searches for a called test. For more information, see Chapter 28, "Calling Tests." • To add a search path, enter the path in the text box, and click **Add Path** +. The path appears in the list box, below the text box. • To delete a search path, select the path and click **Remove Path** ✕. • To move a search path up one position in the list, select the path and click **Move Item Up** ↑. • To move a selected path down one position in the list, select the path and click **Move Item Down** ↓. 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 are preceded by [TD]. Note that you cannot use the **Browse** button to specify search paths in a TestDirector database. **Notes:** 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 44, "Setting Testing Options from a Test Script." You can also set this option using the corresponding *-search_path* command line option, described in Chapter 36, "Running Tests from the Command Line."

Setting Recording Options

The **Record** category contains options for controlling how WinRunner records tests.



Record

Default recording mode: Context Sensitive

☑ Consider child windows

□ Record Start menu by index

□ Record keypad keys as special keys

□ Record shifted keys as upper-case when CAPS LOCK on

☑ Record single-line edit fields as edit_set

□ Record non-unique list items by name

Record owner-drawn buttons as: Object

Maximum length of list item to record: 253

Attached Text

Search radius (in pixels): 35

Preferred search area: Default

In addition to the options in this category, you can set additional recording options in the **Selected Applications**, **Script Format**, and **Text Recognition** sub-categories.

The **Record** category contains the following options:

Option	Description
Default recording mode	Determines the default recording mode—**Context Sensitive** or **Analog**. While you are recording your test, you can switch between recording modes. For more information, see Chapter 3, "Understanding How WinRunner Identifies GUI Objects." **Default** = Context sensitive
Consider child windows	When selected, WinRunner recognizes any MSW_class window, or any object mapped to this class, as a parent object. When cleared, WinRunner recognizes only top-level windows and MDI frames as parent objects. 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 44, "Setting Testing Options from a Test Script." **Default** = selected

Option	Description

Record Start menu by index	Determines how WinRunner records on the Windows **Start** menu in Windows NT.
	When this option is selected, WinRunner records the index IDs for each selected menu item. For example:
	`button_press ("Start");`
	`menu_select_item ("item_2;item_0;item_4");`
	Select this option when the menu item position is constant, but the name of the menu you want to select may change. For example, if the name of the menu option is generated dynamically.
	When this option is cleared, WinRunner records the name of the menu items in the start menu. For example:
	`button_press ("Start");`
	`menu_select_item ("Programs;Accessories;Calculator");`
	Default = **cleared**

Option	Description

Record keypad keys as special keys	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");`
	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.
	Default = cleared

Record shifted keys as uppercase when CAPS LOCK on	Determines whether WinRunner records pressing letter keys and the SHIFT key together as uppercase letters when CAPS LOCK is activated. 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

Option	Description

Record single-line edit fields as edit_set	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");`
	For more information about the **edit_set** and **obj_type** functions, refer to the *TSL Reference*.
	Default = selected

Record non-unique list items by name	Determines whether WinRunner records non-unique ListBox and ComboBox items by name (selected) or by index (cleared). **Notes:** 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 44, "Setting Testing Options from a Test Script." You can also set this option using the corresponding *-rec_item_name* command line option, described in Chapter 36, "Running Tests from the Command Line." **Default** = cleared

Option	Description
Record owner-drawn buttons as	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). 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 44, “Setting Testing Options from a Test Script.” **Default** = Object
Maximum length of list item to record	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 253 characters. **Default** = 253 [characters]
Attached Text	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. 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.

Option	**Description**
Search radius	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. 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 44, "Setting Testing Options from a Test Script." **Default**= 35 [pixels]
Preferred search area	Specifies the location on a GUI object from which WinRunner searches for its attached text. 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 44, "Setting Testing Options from a Test Script." In WinRunner, version 7.01 and earlier, 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. **Default** = Default

Setting Selected Applications

The Selected Applications pane enables you to instruct WinRunner to record your operations on selected programs while ignoring operations on other programs. For example, you may not want to record operations you perform on your e-mail client while recording a test.

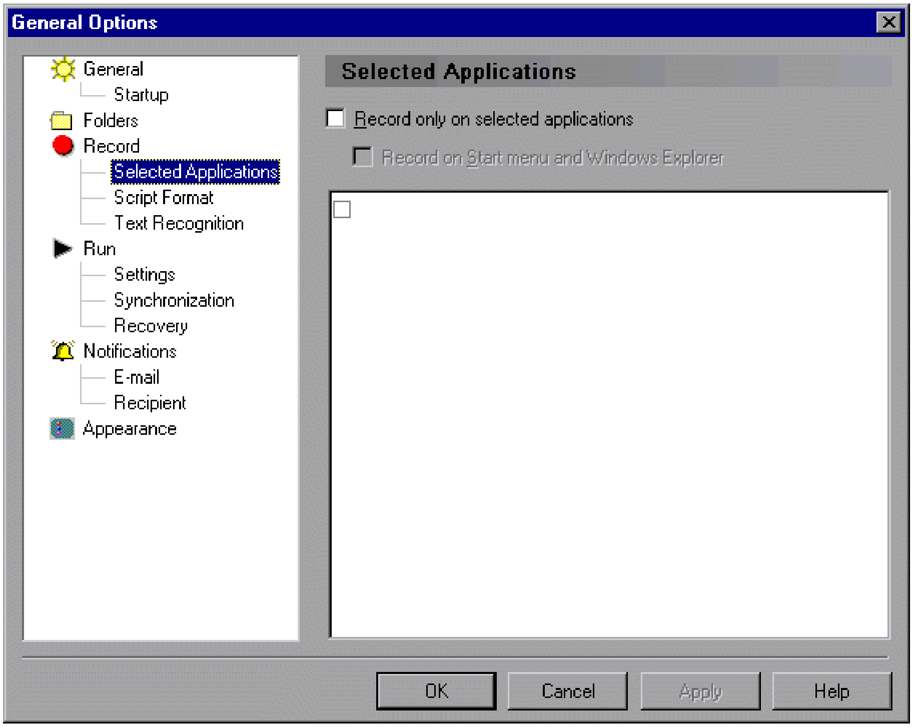
When you enable selective recording, only actions on the selected programs are recorded.

Note that even if you choose only to record on selected applications, you can still create checkpoints and perform all other non-recording WinRunner operations on all applications.

To enable selective recording:

1 Choose **Tools** > **General Options**. The General Options dialog box opens.

2 Click the **Selected Applications** category.



General Options

Selected Applications

☐ Record only on selected applications

☐ Record on Start menu and Windows Explorer

3 Select **Record only on selected applications.**

4 If you want to record operations on the **Start** menu and on Windows Explorer, select **Record on Start menu and Windows Explorer.**

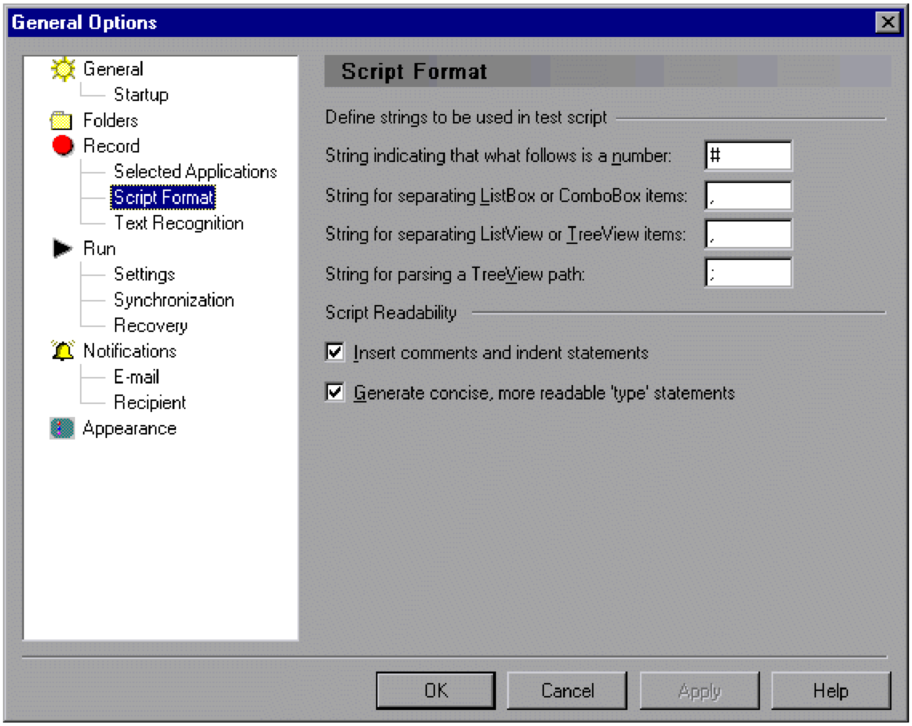
5 If you do not want to record on Internet Explorer and/or Netscape, clear the options for **iexplore.exe**, **netscape.exe**, and/or **netscp6.exe** in the applications list.

6 To add a new application to the list, click an empty list item. Enter the application process file name in the box, or use the browse button to find and select the application process.

Note: Be sure to enter the application process that you want to record. In some cases the process file name is not the same as the name of the file name you use to run the application.

Setting Script Format Options

The **Script Format** category contains options for controlling the appearance and readability of your script.



General Options

Script Format

Define strings to be used in test script

String indicating that what follows is a number: #

String for separating ListBox or ComboBox items: ,

String for separating ListView or TreeView items: ,

String for parsing a TreeView path: ;

Script Readability

☑ Insert comments and indent statements

☑ Generate concise, more readable 'type' statements

The **Script Format** category contains the following options:

Option	Description
String indicating that what follows is a number	The string recorded in the test script to indicate that a list item is specified by its index number. 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 44, “Setting Testing Options from a Test Script.” **Default** = #
String for separating ListBox or ComboBox items	The string recorded in the test script to separate items in a ListBox or a ComboBox. 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 44, “Setting Testing Options from a Test Script.” **Default** = ,
String for separating ListView or TreeView items	The string recorded in the test script to separate items in a ListView or a TreeView. 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 44, “Setting Testing Options from a Test Script.” **Default** = ,
String for parsing a TreeView path	The string recorded in the test script to separate items in a tree view path. 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 44, “Setting Testing Options from a Test Script.” **Default** = ;

Option	Description
Insert comments and indent statements	Determines whether WinRunner automatically divides your test script into sections while you record. For more information, see "Inserting Comments and Indent Statements" below. **Default** = selected

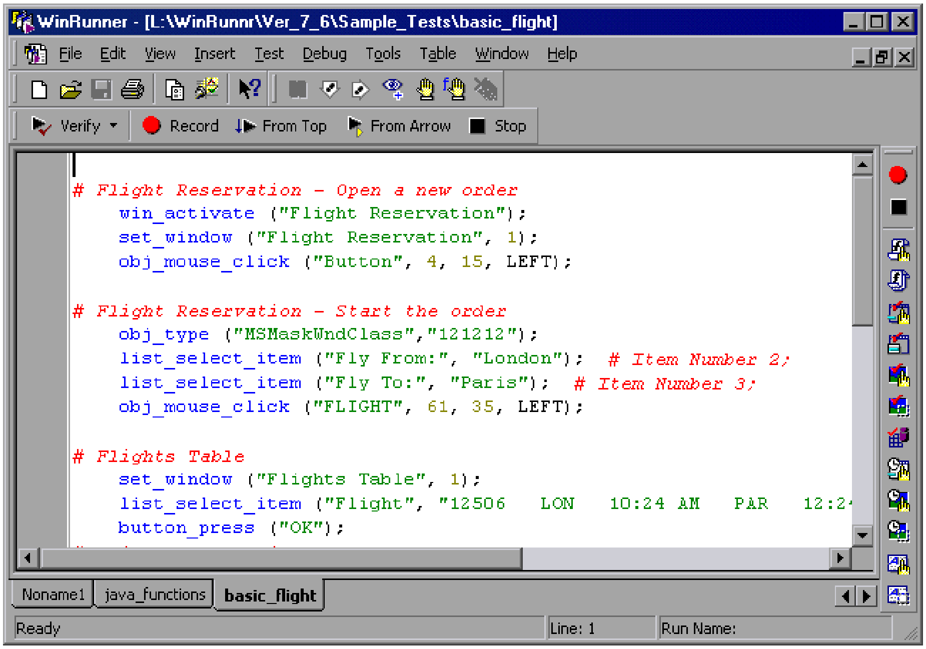
Generate concise, more readable 'type' statements	Determines how WinRunner generates **type**, **win_type**, and **obj_type** statements in a test script.
	When this option is selected, WinRunner generates more concise **type, win_type**, and **obj_type** statements that represent only the net result 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.
	For more information, refer to the **type**, **win_type**, and **obj_type** functions in the *TSL Reference*.
	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 44, "Setting Testing Options from a Test Script."
	Default = selected

Inserting Comments and Indent Statements

When you select the **Insert comments and indent statements** option, WinRunner automatically:

- divides your test script into sections while you record, based on window focus changes.
- inserts comments describing the current window.
- indents the statements under each comment.

This option enables you to easily group all statements related to the same window.


WinRunner - [L:\WinRunnr\Ver_7_6\Sample_Tests\basic_flight]
File Edit View Insert Test Debug Tools Table Window Help
Verify
Record
From Top
From Arrow
Stop
Noname1
java_functions
basic_flight
Ready
Line: 1
Run Name:

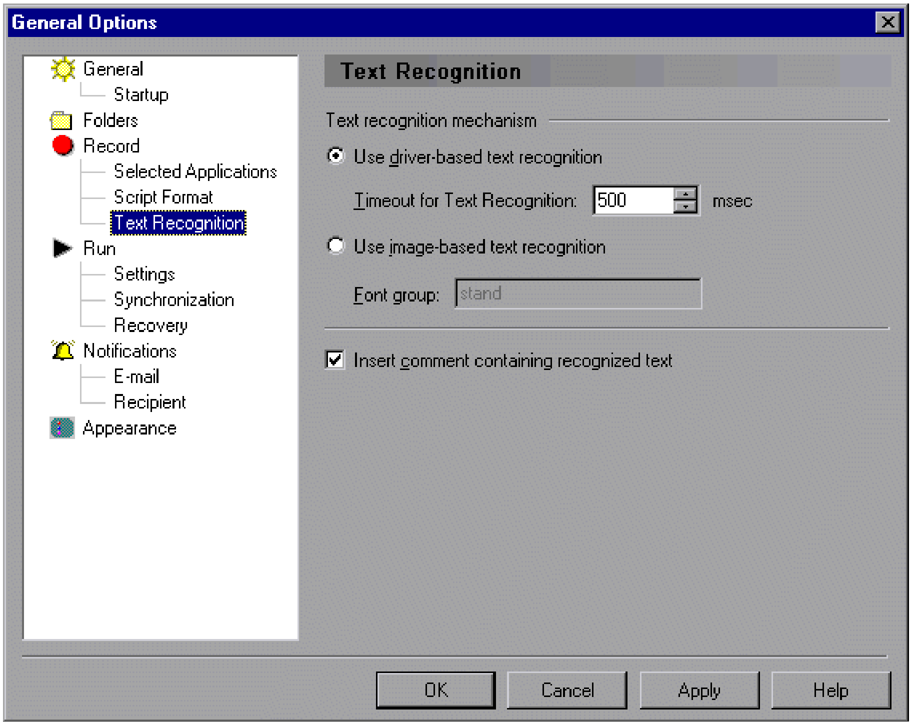
```
# Flight Reservation - Open a new order
    win_activate ("Flight Reservation");
    set_window ("Flight Reservation", 1);
    obj_mouse_click ("Button", 4, 15, LEFT);

# Flight Reservation - Start the order
    obj_type ("MSMaskWndClass","121212");
    list_select_item ("Fly From:", "London");  # Item Number 2;
    list_select_item ("Fly To:", "Paris");  # Item Number 3;
    obj_mouse_click ("FLIGHT", 61, 35, LEFT);

# Flights Table
    set_window ("Flights Table", 1);
    list_select_item ("Flight", "12506   LON   10:24 AM   PAR   12:2
    button_press ("OK");
```

When this option is selected, WinRunner automatically divides your test into sections while you record. A **set_window** statement, as well as any **win_*** statement, can create a division. A new division also begins when you switch from context sensitive to analog recording.

For each new section that WinRunner creates, it inserts a comment with the window name. All of the statements that are recorded while the same window remains in focus are indented under that comment. If you record in Analog mode while this option is selected, the comment is always: Analog Recording.

Setting Text Recognition Options

The **Text Recognition** category options affect how WinRunner recognizes text in your application.



General Options

Text Recognition

Text recognition mechanism

● Use driver-based text recognition

Timeout for Text Recognition: 500 msec

○ Use image-based text recognition

Font group: stand

☑ Insert comment containing recognized text

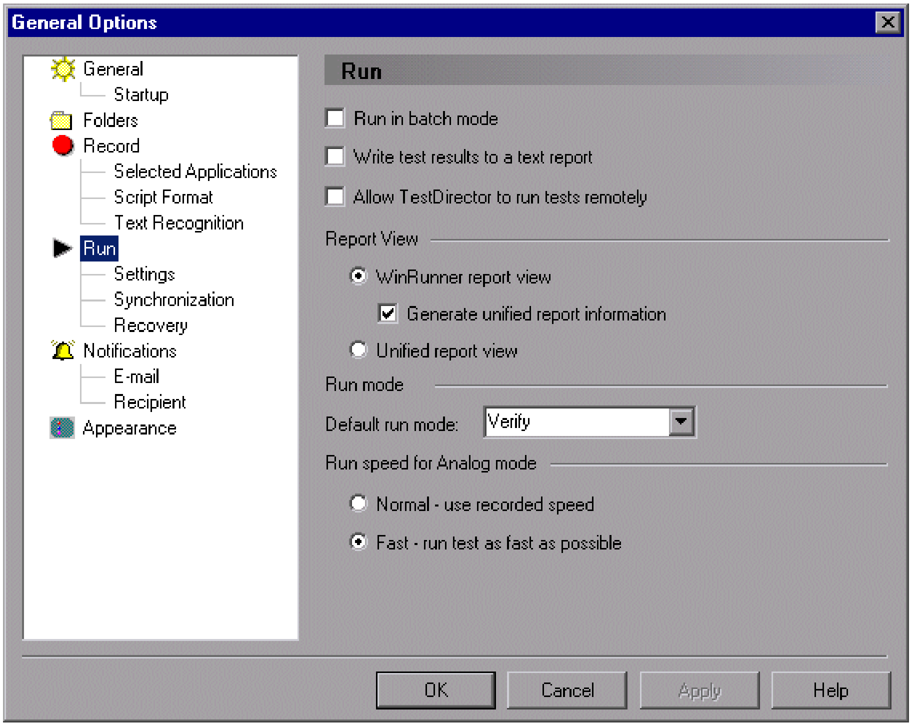
The **Text Recognition** category contains the following options:

Option	Description
Use driver-based text recognition	Uses your graphics driver to recognize text. This method generally yields the most reliable text results. Only if this method does not work well for the application your are testing, select **Use image-based text recognition**. **Default** = selected
Timeout for Text Recognition	Sets the maximum interval (in milliseconds) that WinRunner waits to recognize text when performing a text checkpoint using the driver-based text recognition method during a test run. See "Choosing Appropriate Timeout and Delay Settings" on page 817 for more information on when to adjust this setting. **Default** = 500 [milliseconds]
Use image-based text recognition	Enables WinRunner to recognize text whose font is defined in a font group. Choose this option only if you find that the driver-based text recognition method does not work well with the application you are testing. **Default** = cleared

Option	Description
Font group	Sets the active font group for image text recognition. For more information on font groups, see "Teaching Fonts to WinRunner" on page 395. 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 44, "Setting Testing Options from a Test Script." You can also set this option using the corresponding *-fontgrp* command line option, described in Chapter 36, "Running Tests from the Command Line." **Default** = stand
Insert comment containing recognized text	When you create a text checkpoint, this option determines how WinRunner displays the captured text in the test script. When selected, WinRunner inserts text captured by a text checkpoint during test creation into the test script as a comment. For example, if you choose **Insert** > **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` **Default** = selected

Setting Test Run Options

The **Run** category options control how WinRunner runs tests.



Run

□ Run in batch mode

□ Write test results to a text report

□ Allow TestDirector to run tests remotely

Report View

● WinRunner report view

☑ Generate unified report information

○ Unified report view

Run mode

Default run mode: Verify

Run speed for Analog mode

○ Normal - use recorded speed

● Fast - run test as fast as possible

In addition to the options in this category, you can set additional recording options in the **Settings**, **Synchronization**, and **Recovery** sub-categories.

The **Run** category contains the following options:

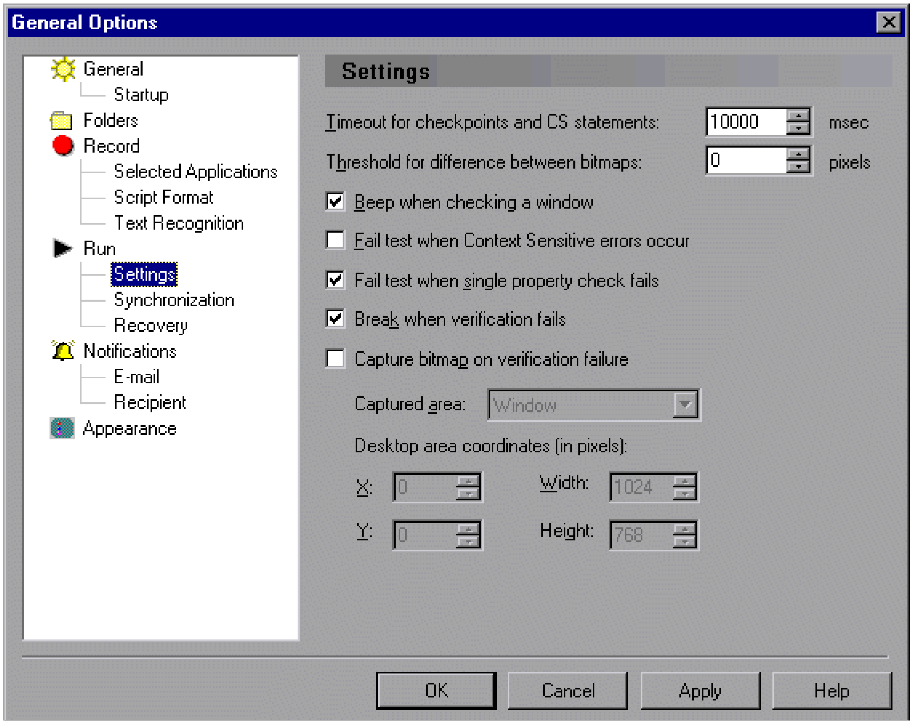
Option	Description
Run in batch mode	Determines whether WinRunner suppresses messages during a test run so that a test can run unattended. For example, if a **set_window** statement is missing from a test script, WinRunner cannot find the specified window. If the test is run in batch mode, WinRunner reports an error in the Test Results window and proceeds to run the next statement in the test script. If 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 selected, WinRunner saves the test results of called tests both under the calling (main batch test) and under the test folder of all first-level called tests. When cleared, the results of all called tests are saved only under the calling test. For more information on suppressing messages during a test run, see Chapter 35, "Running Batch Tests." 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 44, "Setting Testing Options from a Test Script." You can also set this option using the corresponding *-batch* command line option, described in Chapter 36, "Running Tests from the Command Line." **Default** = cleared
Write test results to a text report	Instructs WinRunner to automatically write test results to a text report, called *report.txt*, which is saved in the results folder. You can also set this option using the corresponding *-create_text_report* command line option, described in Chapter 36, "Running Tests from the Command Line." **Note:** A text report of the test results can also be created from the Test Results window (in the WinRunner report view) by choosing **Tools** > **Text Report**. **Default** = cleared

Option	Description
Allow TestDirector to run tests remotely	Enables TestDirector to run WinRunner tests on your machine from a remote machine. For information on running WinRunner tests remotely from TestDirector, refer to your *TestDirector User's Guide*.
WinRunner report view	Displays the test results using the WinRunner test results design. **Default** = selected
Generate unified report information	Generates the necessary information for creating a Unified Report so that you can choose to view the Unified Report of your tests at a later time. (Enabled only when WinRunner report view is selected.) **Default** = selected
Unified report view	Generates unified report information during the test run and displays the test results using the TestFusion unified report design. This design enables you to view all WinRunner events and QuickTest steps in a single report. **Note:** The WinRunner report is always automatically generated when you select this option, enabling you to switch to the WinRunner report view at a later time. **Default** = cleared

Option	Description
Default run mode	Enables you to select the run mode that is used for all tests by default. • **Update**—Used to update the expected results of a test or to create a new expected results folder. • **Verify**—Used to check your application. • **Debug**—Used to help you identify bugs in a test script. For more information on run modes, see "WinRunner Test Run Modes" on page 621. **Default** = Verify
Run speed for Analog mode	Determines the default run speed for tests run in Analog mode. **Normal**—runs the test at the speed at which it was recorded. **Fast**—runs the test as fast as the application can receive input. 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 44, "Setting Testing Options from a Test Script." You can also set this option using the corresponding -*speed* command line option, described in Chapter 36, "Running Tests from the Command Line." **Default** = Fast

Setting Run Setting Options

The **Settings** category contains options for handling specific situations during the test run.



General Options

Settings

Timeout for checkpoints and CS statements: 10000 msec

Threshold for difference between bitmaps: 0 pixels

☑ Beep when checking a window

□ Fail test when Context Sensitive errors occur

☑ Fail test when single property check fails

☑ Break when verification fails

□ Capture bitmap on verification failure

Captured area: Window

Desktop area coordinates (in pixels):

X: 0 Width: 1024

Y: 0 Height: 768

The **Settings** category contains the following options:

Option	Description
Timeout for checkpoints and CS statements	Sets the global timeout (in milliseconds) used by WinRunner when performing checkpoints and Context Sensitive statements. This value is added to the *time* parameter embedded in GUI checkpoint or synchronization point statements to determine the maximum amount of time that WinRunner searches for the specified window or object. The timeout must be greater than the delay for window synchronization (as set in the **Delay for Window Synchronization** option in the **Synchronization** category). For example, when the delay is 2,000 milliseconds and the timeout is 10,000 milliseconds, WinRunner checks the window or object in the application under test every two seconds until the check produces the desired results or until ten seconds have elapsed. **Note:** This option is accurate to within 20-30 milliseconds. See "Choosing Appropriate Timeout and Delay Settings" on page 817 for more information on when to adjust this setting. 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 44, "Setting Testing Options from a Test Script." You can also set this option using the corresponding -*timeout_msec* command line option, described in Chapter 36, "Running Tests from the Command Line." **Default** = 10000 [milliseconds]

Option	Description
Threshold for difference between bitmaps	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. 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 44, "Setting Testing Options from a Test Script." You can also set this option using the corresponding *-min_diff* command line option, described in Chapter 36, "Running Tests from the Command Line." **Default** = 0 [pixels]
Beep when checking a window	Determines whether WinRunner beeps when checking any window during a test run. 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 44, "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 36, "Running Tests from the Command Line." **Default** = selected

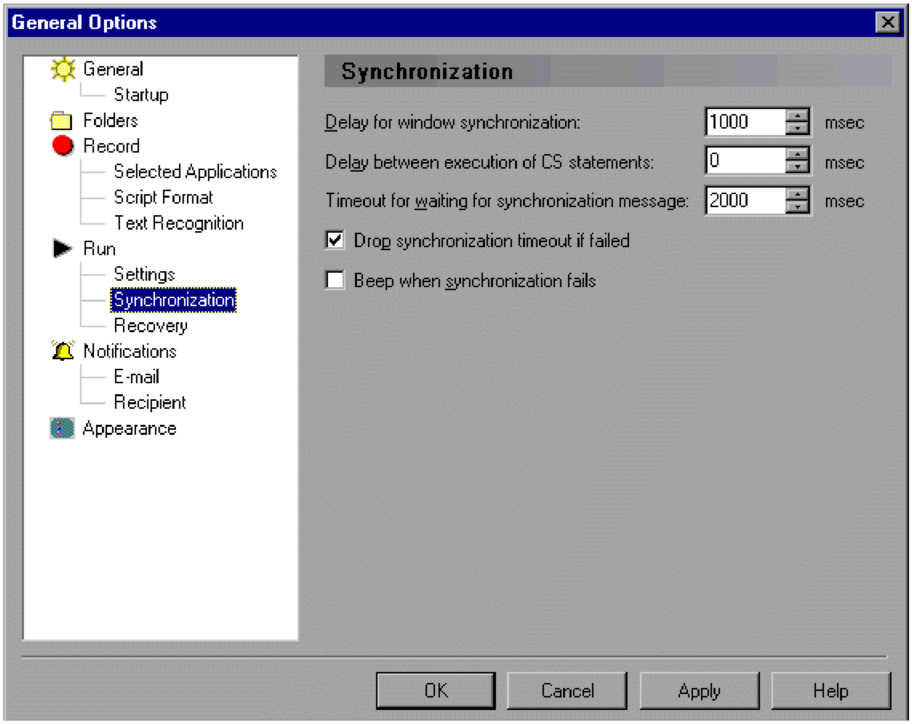
Option	Description
Fail test when Context Sensitive errors occur	Determines whether WinRunner fails a test when Context Sensitive errors occur. A Context Sensitive error is the failure of a Context Sensitive statement during a test run. Context Sensitive errors often occur when WinRunner cannot identify 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 Reference*. 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 44, "Setting Testing Options from a Test Script." You can also set this option using the corresponding *-cs_fail* command line option, described in Chapter 36, "Running Tests from the Command Line." **Default** = cleared
Fail test when single property check fails	Determines whether WinRunner fails a test when **_check_info** statements fail. It also writes an event to the Test Results window for these statements. (You can create **_check_info** statements using the **Insert** > **GUI Checkpoint** > **For Single Property** command.) For information about the **check_info** functions, refer to the *TSL Reference*. 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 44, "Setting Testing Options from a Test Script." You can also set this option using the corresponding *-single_prop_check_fail* command line option, described in Chapter 36, "Running Tests from the Command Line." **Default** = selected

Option	Description
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. This option should be used only when working interactively (not in batch mode). 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. 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 44, “Setting Testing Options from a Test Script.” You can also set this option using the corresponding *-mismatch_break* command line option, described in Chapter 36, “Running Tests from the Command Line.” **Default** = selected
Capture bitmap on verification failure	Instructs WinRunner to capture an image of your application each time a checkpoint fails. The bitmap is saved in your test results folder. **Default** = cleared You can use the **setvar** and **getvar** functions to set and retrieve the value of the corresponding **capture_bitmap** testing option from within a test script, as described in Chapter 44, “Setting Testing Options from a Test Script.”
Captured area	Specifies the area of your screen to capture when a checkpoint fails. **Window**—Captures the active window. **Desktop**—Captures the entire desktop. **Desktop Area**—Captures the specified area of the desktop.

Option	Description
Desktop area coordinates	**X**—The x-coordinate of the top, left corner of the rectangle area to capture. **Y**—The y-coordinate of the top, left corner of the rectangle area to capture. **Width**—The width of the rectangle to capture. **Height**—The height of the rectangle to capture. (Enabled only when **Desktop Area** is the selected **Captured area**.)

Setting Run Synchronization Options

The Synchronization category defines synchronization settings for your test run.



Delay for window synchronization: 1000 msec

Delay between execution of CS statements: 0 msec

Timeout for waiting for synchronization message: 2000 msec

☑ Drop synchronization timeout if failed

☐ Beep when synchronization fails

The **Synchronization** category contains the following options:

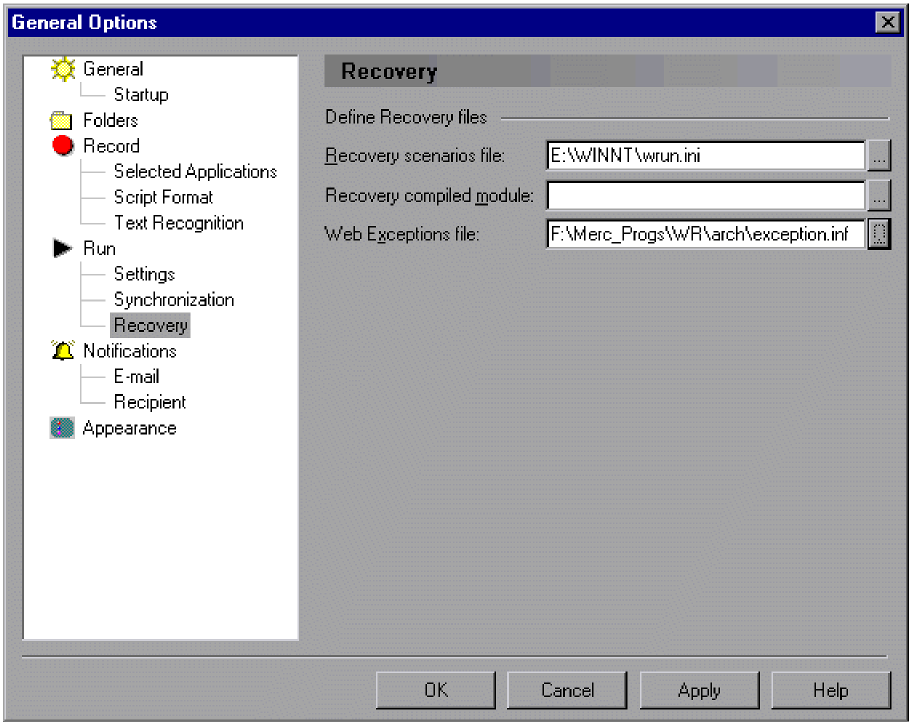
Option	Description
Delay for window synchronization	Sets the sampling interval (in milliseconds) used to determine that a window is stable before capturing it for a Context Sensitive checkpoint or synchronization point. To be declared stable, a window must not change between two consecutive samplings. This sampling continues until the window is stable or the timeout (as set in the **Timeout for Checkpoints and CS Statements** in the **Settings** category) is reached. In general, a smaller delay enables WinRunner to capture the object or window more quickly so that the test can continue, but smaller delays increase the load on the system. This option is accurate to within 20-30 milliseconds. See "Choosing Appropriate Timeout and Delay Settings" on page 817 for more information on when to adjust this setting. 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 44, "Setting Testing Options from a Test Script." You can also set this option using the corresponding -*delay_msec* command line option, described in Chapter 36, "Running Tests from the Command Line." **Default** = 1000 [milliseconds]
Delay between execution of CS statements	Sets the time (in milliseconds) that WinRunner waits before executing each Context Sensitive statement when running a test. See "Choosing Appropriate Timeout and Delay Settings" on page 817 for more information on when to adjust this setting. 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 44, "Setting Testing Options from a Test Script." You can also set this option using the corresponding -*cs_run_delay* command line option, described in Chapter 36, "Running Tests from the Command Line." **Default** = 0 [milliseconds]

Option	Description
Timeout for waiting for synchronization message	Sets the timeout (in milliseconds) that WinRunner waits before validating that keyboard or mouse input was entered correctly during a test run. If synchronization often fails during your test runs, consider increasing the value of this option. See "Choosing Appropriate Timeout and Delay Settings" on page 817 for more information on when to adjust this setting. 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 44, "Setting Testing Options from a Test Script." **Default** = 2000 [milliseconds]
Drop synchronization timeout if failed	Determines whether WinRunner minimizes the synchronization timeout (as defined in the **Timeout for Waiting for Synchronization Message** option above) after the first synchronization failure. See "Choosing Appropriate Timeout and Delay Settings" on page 817 for more information on when to adjust this setting. 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 44, "Setting Testing Options from a Test Script." **Default** = selected

Option	Description
Beep when synchronization fails	Determines whether WinRunner beeps when the timeout for waiting for synchronization message fails. This option is primarily for debugging test scripts. 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. See “Choosing Appropriate Timeout and Delay Settings” on page 817 for more information on when to adjust this setting. 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 44, “Setting Testing Options from a Test Script.” **Default** = cleared

Setting Recovery Options

The Recovery category options specify the files that WinRunner refers to for recovery scenario and Web exception information.



Define Recovery files

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

Recovery compiled module:

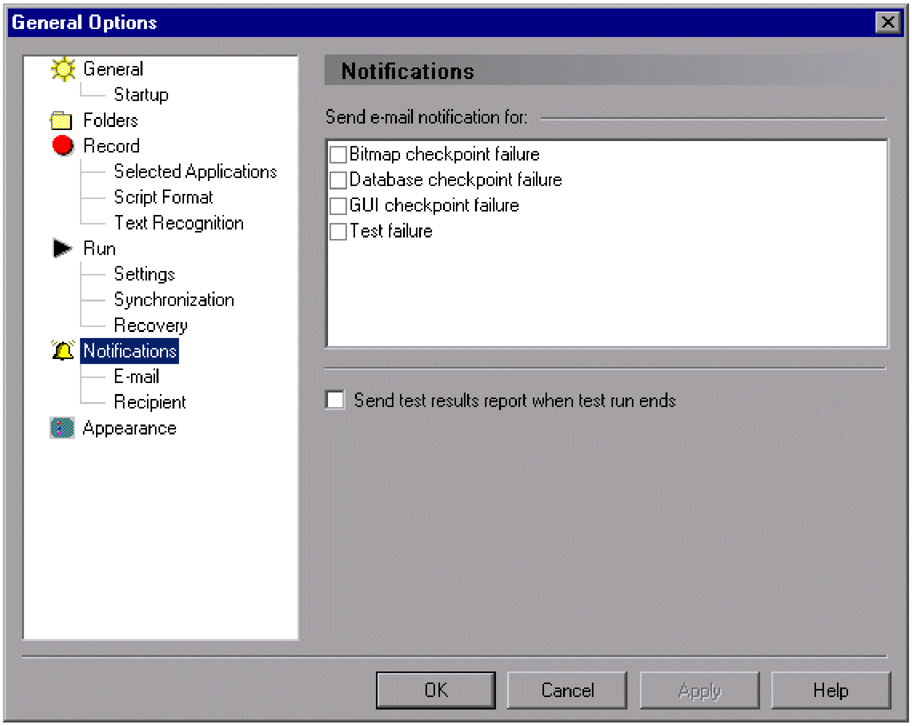
Web Exceptions file: F:\Merc_Progs\WR\arch\exception.inf

The **Recovery** category contains the following options:

Option	Description
Recovery scenario file	Indicates the location of the recovery scenarios file, which stores the details of the available recovery scenarios. You must select a recovery scenarios file other than *wrun.ini* before you can use the Recovery Manager to create or modify recovery scenarios. Recovery scenarios are defined and modified in the Recovery Manager. For more information, see Chapter 23, "Defining and Using Recovery Scenarios." **Default = <Windows folder>\wrun.ini**
Recovery compiled module	Indicates the location of the exceptions compiled module, which is loaded automatically when WinRunner opens, and contains the recovery and post-recovery functions used in recovery scenarios. Enter a new module name, or enter the name of an existing compiled module. For more information, see Chapter 23, "Defining and Using Recovery Scenarios." **Note:** You can specify a TestDirector script as your recovery compiled module. If you do, ensure that **Reconnect on startup** is selected in the TestDirector Connection dialog box. For more information, see See "Connecting to and Disconnecting from a Project" on page 917.
Web Exceptions file	Indicates the location of the Web exceptions file, which stores the details of the available Web exception handling definitions. Web exceptions are defined and modified in the Web Exception Editor. For more information, see Chapter 24, "Handling Web Exceptions." **Default = <WinRunner installation folder>\arch\exception.inf**

Setting Notification Options

The **Notifications** category contains options for sending e-mail notifications based on specified criteria.



Send e-mail notification for:

□ Bitmap checkpoint failure
□ Database checkpoint failure
□ GUI checkpoint failure
□ Test failure

□ Send test results report when test run ends

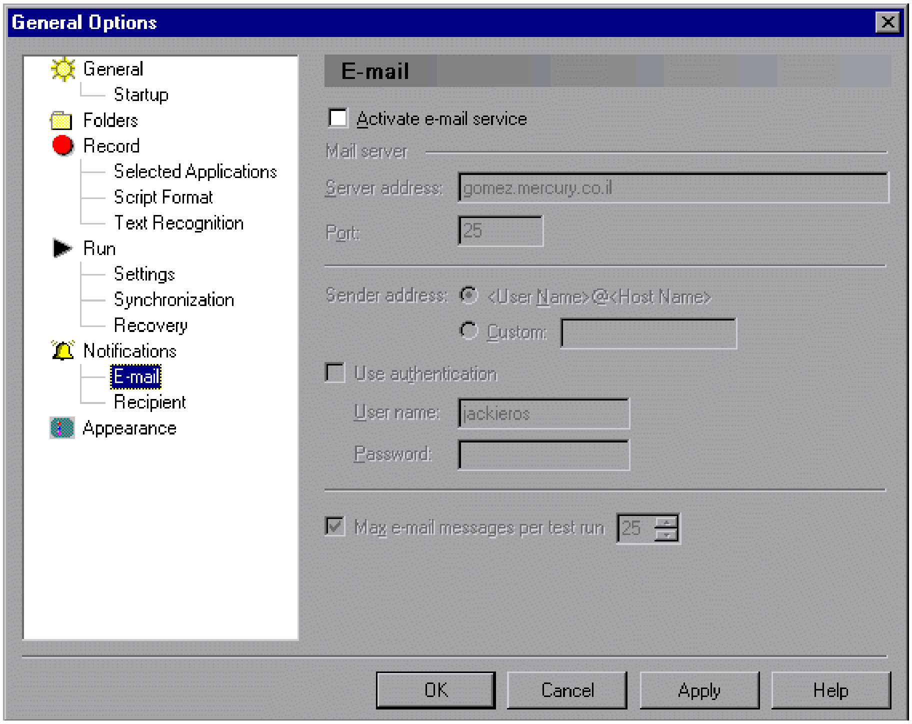
In addition to the options in this category, you can set additional notification options in the **E-mail** and **Recipient** subcategories.

The **Notifications** category contains the following options:

Option	Description
Send e-mail notification for	Sends an e-mail for the selected conditions. You can select one or more of the following: • **Bitmap checkpoint failure**—Sends an e-mail to the specified Recipients each time a bitmap checkpoint fails. The e-mail contains summary details about the test, the checkpoint, and the file names for the expected, actual, and difference images. • **Database checkpoint failure**—Sends an e-mail to the specified Recipients each time a database checkpoint fails. The e-mail contains summary details about the test, the checkpoint, and details about the connection string and SQL query used for the checkpoint. • **GUI checkpoint failure**—Sends an e-mail to the specified Recipients each time a GUI checkpoint fails. The e-mail contains summary details about the test, the checkpoint, and details about the expected and actual values of the property check. • **Test failure**— Sends an e-mail to the specified Recipients each time a test run fails. The e-mail contains the summary test results in text format. **Default** = all check boxes are cleared
Send test results report when test run ends	Sends an e-mail to the specified recipients (see **Recipients** category) at the end of each test run. The e-mail contains the summary test results in text format. **Notes:** If you also select to send e-mail notifications for **Test failure**, and the test run fails, then only the Test failure e-mail is sent. To enable the notification options selected in this pane, you must select to **Activate e-mail service** option in the **E-mail** category. **Default** = Selected

Setting E-mail Notification Options

The **E-mail** category contains options for specifying the mail server to use and other e-mail preferences.



General Options

E-mail

□ Activate e-mail service

Mail server

Server address: gomez.mercury.co.il

Port: 25

Sender address: ● <User Name>@<Host Name>

○ Custom:

□ Use authentication

User name: jackieros

Password:

☑ Max e-mail messages per test run 25

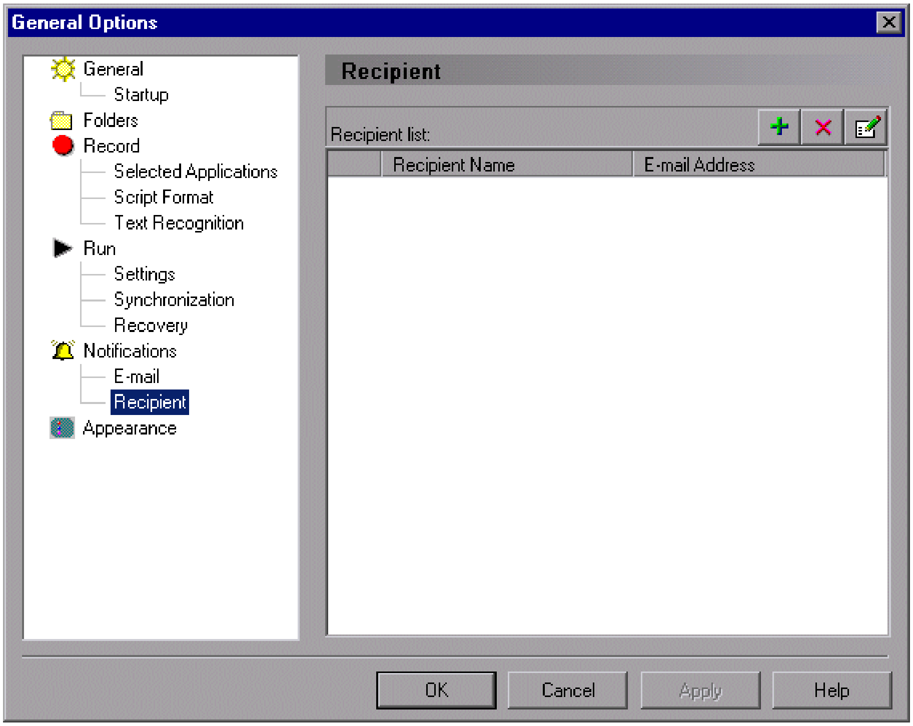
The **E-mail** category contains the following options:

Option	Description
Activate e-mail service	Instructs WinRunner to enable the e-mail notification options that are set in the **Notifications** category as well as any specified in the test script using the **email_send_msg** function. You can use the **setvar** and **getvar** functions to set and retrieve the value of the corresponding **email_service** testing option from within a test script, as described in Chapter 44, "Setting Testing Options from a Test Script." **Default** = cleared
Server address	The address of the outgoing mail server you want to use to send the e-mail message.
Port	The mail server port to use. **Default** = 25
Sender address	The e-mail address you want to display as the sender of the e-mail notification. Choose one of the following: • **<User Name>@<Host Name>**—Uses the login name and host name of the WinRunner computer on which the test was run as the sender address. For example: Amy@MYCOMPUTER • **Custom**—Enables you to specify any text or e-mail address as the sender address. **Note:** Many mail servers require that the sender name is a valid e-mail address. If the outgoing mail server you specified has such a requirement, use the Custom option to specify a valid e-mail address. If you do not specify a valid e-mail address for such a server, WinRunner sends the e-mail to the mail server, but the mail server will not send the e-mail to the recipients. **Default** = <User Name>@<Host Name>

Option	**Description**
Use authentication	Indicates that your outgoing mail server requires you to log on in order to send e-mail. When this option is selected, you must enter the log on user name and password. **Default** = cleared
Maximum e-mail notifications per test run	The maximum number of e-mail notifications you want to send to the recipients (as specified in the **Recipients** category) during a test run. **Note:** This option applies only to the number of e-mail messages that WinRunner sends according to the options set in the **Notifications** category. Messages sent using the **email_send_msg** function are completely independent of this option. For more information on the **email_send_msg** function, refer to the *TSL Reference*. **Default** = 25

Setting Notification Recipients Options

The **Recipients** category enables you to specify the recipients to receive e-mail notifications.



➤ Click **Add Recipient** + to add a new recipient to the list.

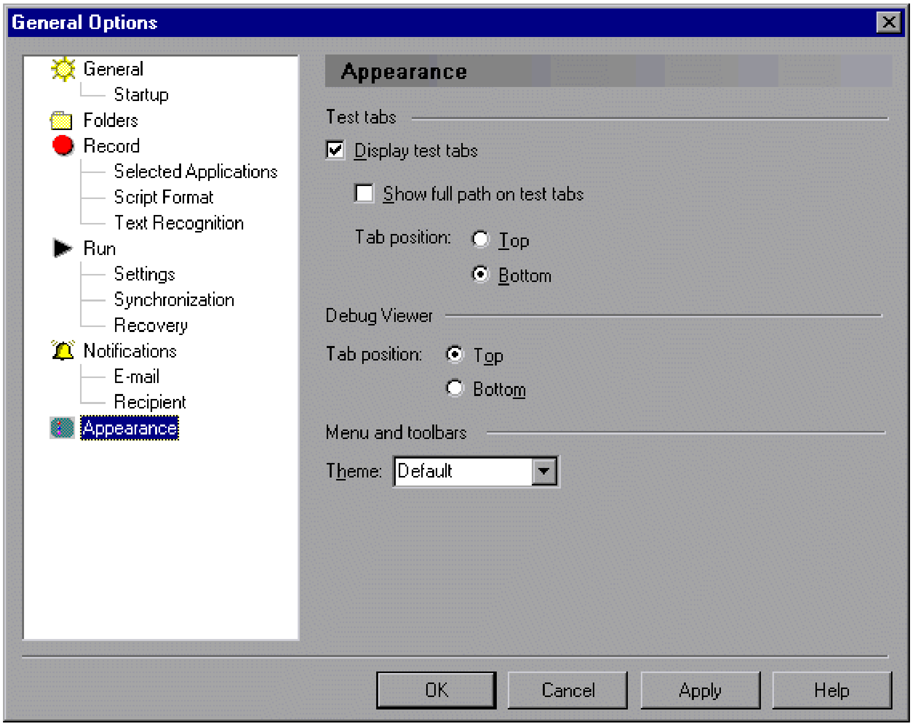
➤ Select a recipient from the list and click **Remove Recipient** ✕ to remove the recipient from the list.

➤ Select a recipient from the list and click **Modify Recipient Details** ✎ to modify the details of a recipient in the list.

Note: Some mail servers (such as Microsoft Exchange, if configured to do so) prevent mail clients other than Microsoft Outlook from sending e-mail outside the organization. If the outgoing mail server you specified in the **E-mail** category has configured such a limitation, confirm that you specify only e-mail addresses with a domain name that matches your mail server's domain name. If you specify external recipients, the WinRunner mail client sends the e-mail message to the mail server, but the mail server will not send the message to the recipients. In most cases, the mail server does not send an error message to the sender in these situations.

Setting Appearance Options

The **Appearance** category contains options for controlling the appearance of WinRunner.



General Options

Appearance

Test tabs

☑ Display test tabs

□ Show full path on test tabs

Tab position: ○ Top ● Bottom

Debug Viewer

Tab position: ● Top ○ Bottom

Menu and toolbars

Theme: Default

The **Appearance** category contains the following options:

Option	Description
Display test tabs	Displays a tab for each open test so that you can display an open test by clicking its tab. If this option is cleared, you can select a test to display using the **Window** menu commands. **Default** = selected
Show full path on test tabs	When this option is selected, displays the full path of the test on each test tab. When this option is cleared, only the test name is displayed on the tab. **Default** = cleared
Tab position **(Test tabs)**	Indicates whether to display the test tabs at the **Top** or **Bottom** of the page.
Tab position **(Debug Viewer)**	Indicates whether to display the debug tabs at the **Top** or **Bottom** of the Debug Viewer window.
Theme	Enables you to select a pre-configured style for your menu and toolbar items.

Choosing Appropriate Timeout and Delay Settings

The table below summarizes the timeout and delay settings available in the General Options dialog box, and describes the situations in which you may want to adjust each setting.

Setting	Description	Adjustment Recommendations	Default
Delay for Window Synchronization	The amount of time WinRunner waits between each attempt to locate a window or object - enabled window to stabilize.	A smaller delay enables WinRunner to capture the object or window more quickly so that the test can continue, but smaller delays increase the load on the system. In most cases, when you modify the Timeout for Checkpoints and CS Statements, you should modify the delay in order to maintain a constant ratio. To avoid overloading your system, you should not exceed a timeout:delay ratio of 50:1.	1000 (ms)
Timeout for checkpoint and CS statements	The amount of time, in addition to the time parameter embedded in a GUI checkpoint or synchronization point, that WinRunner waits for an object or window to appear.	You should increase this setting if your application takes longer than the current timeout value to successfully display objects and windows. If only one or few objects have this problem, however, it may be preferable to add a synchronization point to the script for the problematic objects.	10000 (ms)

Setting	Description	Adjustment Recommendations	Default
Delay between execution of CS statements	Amount of time WinRunner waits before executing each CS statement.	Increase this delay when you need to slow down the test run for reasons not related to synchronization issues. For example, you may want to increase the delay so that you can follow the test as it runs step by step.	0 (ms)
Timeout for waiting for synchronization message	The amount of time WinRunner waits before validating that keyboard or mouse input was entered correctly during a test run.	Increase this setting if WinRunner runs the script faster than the application is capable of executing the statements.	2000 (ms)
Drop synchronization timeout if failed	Automatically minimizes the length of the **Timeout for waiting for synchronization message** setting after the first synchronization validation failure. This increases the likelihood that the test will fail quickly, as mouse and keyboard entries will not be complete.	Select this option to prevent the test from running for a long time with incorrect data due to an incomplete mouse or keyboard entry.	selected

Setting	Description	Adjustment Recommendations	Default
Beep when synchronization fails	WinRunner beeps each time the **Timeout for waiting for synchronization message** setting is exceeded.	You may want to select this option while debugging your script. If you hear many beeps during a single test run, increase the **Timeout for waiting for synchronization message**.	cleared
Timeout for text recognition	The amount of time that WinRunner waits to recognize text when performing a text checkpoint using the standard Text Recognition method during a test run.	If text checkpoints fail using the standard Text Recognition method, try increasing this timeout. (Alternatively you can try using Image Text Recognition.)	500 (ms)

42

Customizing the Test Script Editor

WinRunner includes a powerful and customizable script editor. This enables you to set the size of margins in test windows, change the way the elements of a test script appear, and create a list of typing errors that will be automatically corrected by WinRunner.

This chapter describes:

- ➤ About Customizing the Test Script Editor
- ➤ Setting Display Options
- ➤ Personalizing Editing Commands

About Customizing the Test Script Editor

WinRunner's script editor lets you set display options, and personalize script editing commands.

Setting Display Options

Display options let you configure WinRunner's test windows and how your test scripts 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

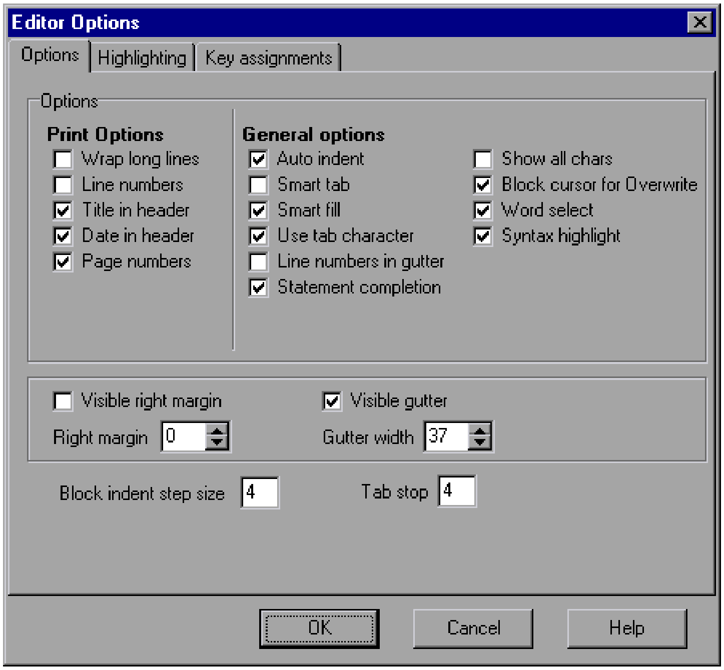
WinRunner's display options let you control how test scripts appear in test windows, how different elements of test scripts are displayed, and how test scripts will appear when they are printed.

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.

To customize the appearance of your script:

1 Choose **Tools** > **Editor Options**. The Editor Options dialog box opens.



Options

Print Options

□ Wrap long lines
□ Line numbers
☑ Title in header
☑ Date in header
☑ Page numbers

General options

☑ Auto indent
□ Smart tab
☑ Smart fill
☑ Use tab character
□ Line numbers in gutter
☑ Statement completion

□ Show all chars
☑ Block cursor for Overwrite
☑ Word select
☑ Syntax highlight

□ Visible right margin
Right margin 0

☑ Visible gutter
Gutter width 37

Block indent step size 4
Tab stop 4

2 Click the **Options** tab.

3 Under the **General options** choose from the following options:

Options	**Description**
Auto indent	Causes lines following an indented line to automatically begin at the same point as the previous line. You can click the Home key on your keyboard to move the cursor back to the left margin.
Smart tab	A single press of the tab key will insert the appropriate number of tabs and spaces in order to align the cursor with the text in the line above.

Options	Description
Smart fill	Insert the appropriate number of tabs and spaces in order to apply the Auto indent option. When this option is not selected, only spaces are used to apply the Auto indent. (Both **Auto indent** and **Use tab character** must be selected to apply this option).
Use tab character	Inserts a tab character when the tab key on the keyboard is used. When this option is not enabled, the appropriate number of space characters will be inserted instead.
Line numbers in gutter	Displays a line number next to each line in the script. The line number is displayed in the test script window's gutter.
Statement completion	Opens a list box displaying all available matches to the function prefix whenever the user presses the CTRL and SPACE keys simultaneously or 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 827.
Visible right margin	Displays a line that indicates the test window's right margin.

Options	Description
Right margin	Sets the position, in characters, of the test window's right margin (0=left window edge).
Visible gutter	Displays a blank area (gutter) in the test window's left margin.
Gutter width	Sets the width, in pixels, of the gutter.
Block indent step size	Sets the number characters that the selected block of TSL statements will be moved (indented) when the INDENT SELECTED BLOCK softkey is used. For more information on editor softkeys, see "Personalizing Editing Commands," on page 829.
Tab stop	Sets the distance, in characters, between each tab stop.

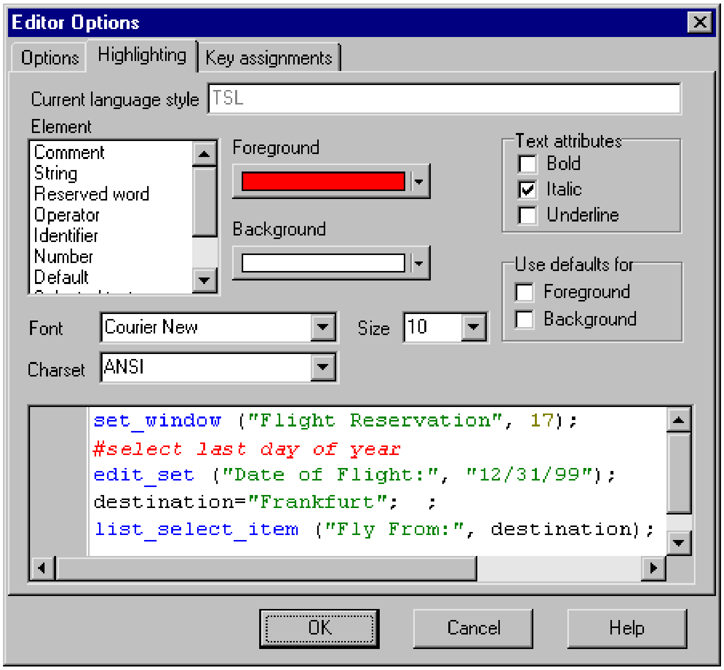
Highlighting Script Elements

WinRunner scripts contain many different elements, such as comments, strings, WinRunner reserved words, operators and numbers. Each element of a WinRunner script is displayed in a different color and style. You can create your own personalized color scheme and style for each script element. For example, all comments in your scripts could be displayed as italicized, blue letters on a yellow background.

To edit script elements:

1 Choose **Tools** > **Editor Options**. The Editor Options dialog box opens.

2 Click the **Highlighting** tab.



Current language style TSL

☐ Bold
☑ Italic
☐ Underline

Use defaults for: ☐ Foreground ☐ Background

Font Courier New   Size 10

Charset ANSI

```
set_window ("Flight Reservation", 17);
#select last day of year
edit_set ("Date of Flight:", "12/31/99");
destination="Frankfurt";  ;
list_select_item ("Fly From:", destination);
```

3 Select a script element from the **Element** list.

4 Choose from the following options:

Options	Description
Foreground	Sets the color applied to the text of the script element.
Background	Sets the color that appears behind the script element.
Text Attributes	Sets the text attributes applied to the script element. You can select bold, italic, or underline or a combination of these attributes.
Use defaults for	Applies the font and colors of the "default" style to the selected style.
Font	Sets the typeface of all script elements.

Options	Description
Size	Set the size, in points, of all script elements.
Charset	Sets the character subset of the selected font.

An example of each change you apply will be displayed in the pane at the bottom of the dialog box.

5 Click **OK** to apply the changes.

Reserved Words

WinRunner contains “reserved words,” which include the names of all TSL functions and language keywords, such as auto, break, char, close, continue, int, function. For a complete list of all reserved words in WinRunner, refer to the *TSL Reference*. You can add your own reserved words in the *[ct_KEYWORD_USER]* section of the *reserved_words.ini* file, which is located in the *dat* folder in the WinRunner installation directory. Use a text editor, such as Notepad, to open the file. Note that after editing the list, you must restart WinRunner so that it will read from the updated list.

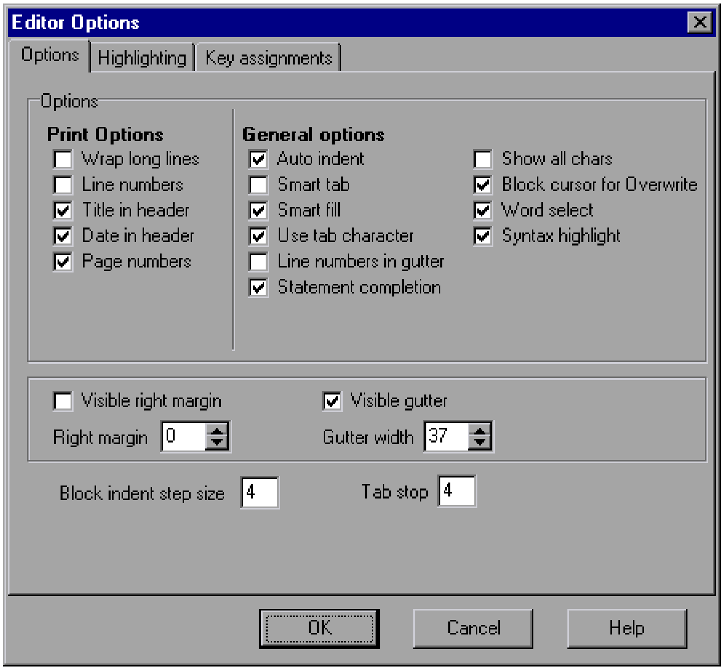
Customizing Print Options

You can set how the hard copy of your script will appear when it is sent to the printer. For example, your printed script can include line numbers, the name of the file, and the date it was printed.

To customize your print options:

1 Choose **Tools** > **Editor Options**. The Editor Options dialog box opens.

2 Click the **Options** tab.



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

□ Visible right margin
☑ Visible gutter

Right margin: 0
Gutter width: 37

Block indent step size: 4
Tab stop: 4

3 Choose from the following Print options:

Option	Description
Wrap long lines	Automatically wraps a line of text to the next line if it is wider than the current printer page settings.
Line numbers	Prints a line number next to each line in the script.
Title in header	Inserts the file name into the header of the printed script.
Date in header	Inserts today's date into the header of the printed script.
Page numbers	Numbers each page of the script.

4 Click **OK** to apply the changes.

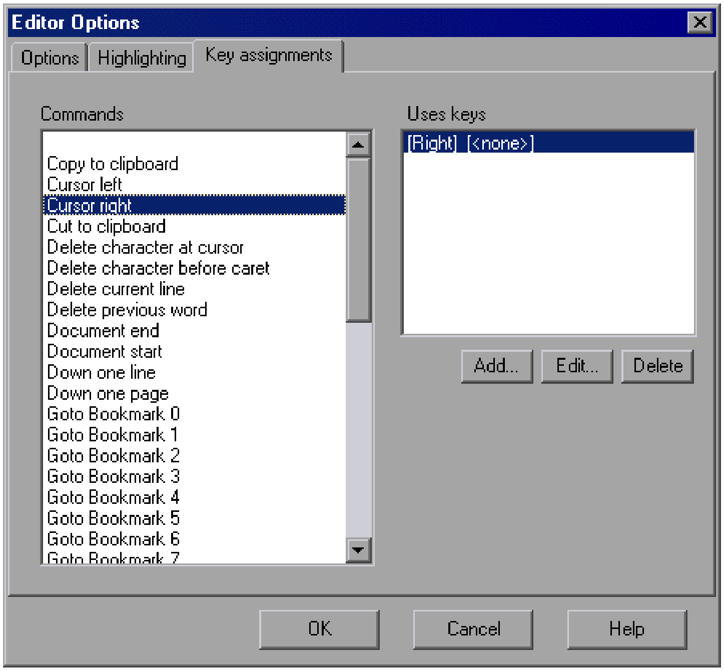
Personalizing Editing Commands

You can personalize the default keyboard commands you use for editing test scripts. WinRunner includes keyboard commands that let you move the cursor, delete characters, cut, copy, and paste information to and from the clipboard. You can replace these commands with your own preferred commands. For example, you could change the Paste command from the default Ctrl + V to Ctrl + P.

To personalize editing commands:

1 Choose **Tools** > **Editor Options**. The Editor Options dialog box opens.

2 Click the **Key assignments** tab.



Commands

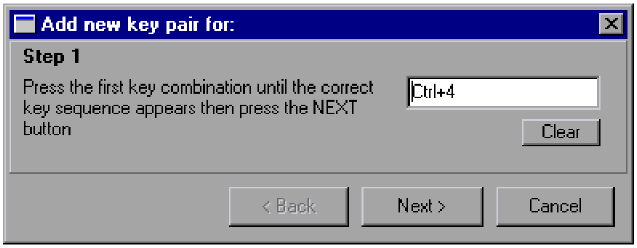
- Copy to clipboard
- Cursor left
- Cursor right
- Cut to clipboard
- Delete character at cursor
- Delete character before caret
- Delete current line
- Delete previous word
- Document end
- Document start
- Down one line
- Down one page
- Goto Bookmark 0
- Goto Bookmark 1
- Goto Bookmark 2
- Goto Bookmark 3
- Goto Bookmark 4
- Goto Bookmark 5
- Goto Bookmark 6
- Goto Bookmark 7

Uses keys

[Right] [<none>]

3 Select a command from the **Commands** list.

4 Click **Add** to create an additional key assignment or click **Edit** to modify the existing assignment. The Add/Edit key pair for dialog box opens. Press the keys you want to use, for example, CTRL + 4:

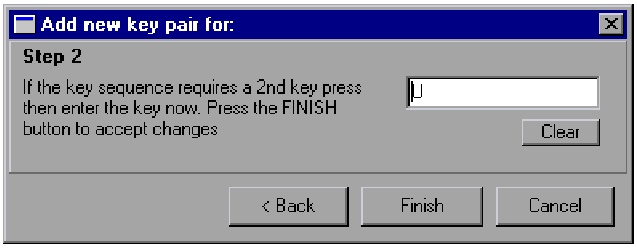

Add new key pair for:

Step 1

Press the first key combination until the correct key sequence appears then press the NEXT button

Ctrl+4

5 Click **Next**. To add an additional key sequence, press the keys you want to use, for example U:


Add new key pair for:

Step 2

If the key sequence requires a 2nd key press then enter the key now. Press the FINISH button to accept changes

U

6 Click **Finish** to add the key sequence(s) to the **Use keys** list.

If you want to delete a key sequence from the list, highlight the keys in the **Uses keys** list and click **Delete**.

7 Click **OK** to apply the changes.

43

Customizing the WinRunner User Interface

You can customize the WinRunner user interface to adapt it to your testing needs and to the application you are testing.

This chapter describes:

- About Customizing WinRunner's User Interface
- Customizing the File, Debug, and User-Defined Toolbars
- Customizing the User Toolbar
- Using the User Toolbar
- Configuring WinRunner Softkeys

About Customizing WinRunner's User Interface

You can adapt WinRunner's user interface to your testing needs by changing the way you access WinRunner commands.

You may find that when you create and run tests, you frequently use the same WinRunner menu commands and insert the same TSL statements into your test scripts. You can create shortcuts to these commands and TSL statements by customizing the WinRunner toolbars.

The application you are testing may use softkeys that are preconfigured for WinRunner commands. If so, you can adapt the WinRunner user interface to this application by using the WinRunner Softkey utility to reconfigure the conflicting WinRunner softkeys.

Customizing the File, Debug, and User-Defined Toolbars

You can use the Customize Toolbars option to create user-defined toolbars and to customize the appearance and contents of the File, Debug, and user-defined toolbars.

Note: You can also customize the User toolbar. For more information, see "Customizing the User Toolbar" on page 839.

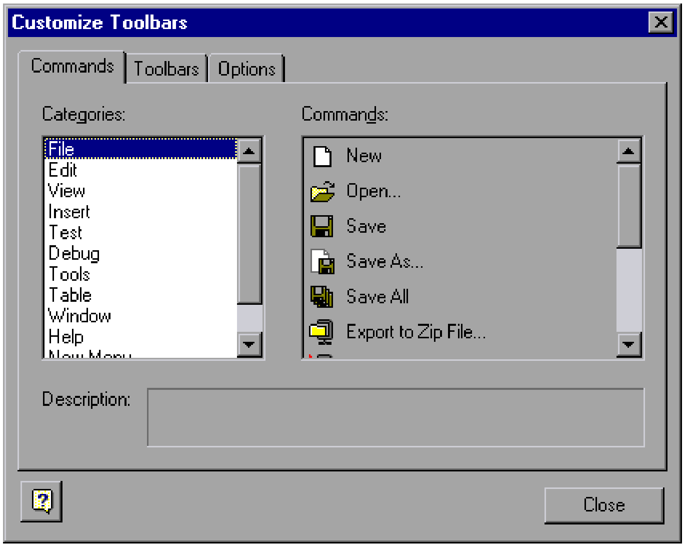
Adding or Removing Toolbar Buttons that Perform Menu Commands

Using the **Commands** tab of the Customize Toolbars dialog box, you can add toolbar buttons that perform frequently-used menu commands to the File and Debug toolbars or to any existing user-defined toolbars. You can also remove toolbar buttons from any of these toolbars.

Tip: You can restore the default buttons to a selected toolbar or to all toolbars using the **Reset** or **Reset All** buttons in the **Toolbars** tab. For more information, see "Controlling the Toolbars Display" on page 834.

To add a button to the File, Debug, or User-Defined Toolbars:

1 Choose **View** > **Customize Toolbars**. The Customize Toolbars dialog box opens and displays the **Commands** tab.



2 In the **Categories** list, find and select the menu name that contains the command you want to add to the toolbar.

3 In the **Commands** list, select the command you want to add and drag it to the File, Debug, or User-Defined toolbar.

4 When you place the button over one of these toolbars, the mouse pointer becomes an I-beam cursor, indicating the location where the button will be placed. Drag the I-beam cursor to the location where you want to add the button, and release the mouse button.

Tip: You can also drag toolbar buttons from one toolbar to another toolbar while the Customize Toolbars dialog box is open.

To remove a button from the File, Debug, or User-Defined Toolbars:

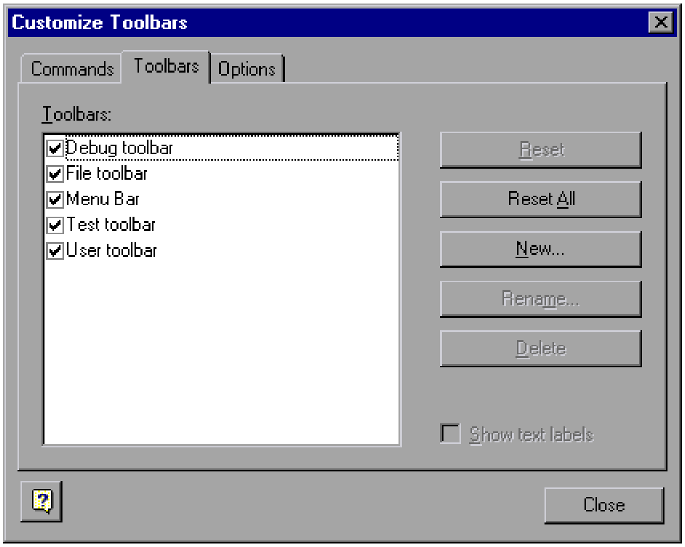
1 Choose **View** > **Customize Toolbars**. The Customize Toolbars dialog box opens.

2 Drag the toolbar button you want to remove from the toolbar to any location outside the toolbars area.

Controlling the Toolbars Display

The **Toolbars** tab of the Customize Toolbars dialog box enables you to display or hide toolbars; restore the default buttons on toolbars; create, rename, and delete user-defined toolbars; and control the appearance of individual toolbars.

Tip: You can also display or hide WinRunner toolbars using the appropriate option in the **View** menu.



To display or hide a toolbar:

1 Choose **View** > **Customize Toolbars**. The Customize Toolbars dialog box opens and displays the **Commands** tab.

2 Click the **Toolbars** tab.

3 Select or clear the check box next to a WinRunner or user-defined toolbar to display or hide it.

Note: You cannot hide the **Menu** bar.

To restore the default buttons on one or all WinRunner toolbars:

1 Choose **View** > **Customize Toolbars**. The Customize Toolbars dialog box opens and displays the **Commands** tab.

2 Click the **Toolbars** tab.

3 To restore the default buttons for a specific toolbar, select the toolbar from the toolbars list and click **Reset**.

Note: The **Reset** button is disabled if a user-defined toolbar is selected.

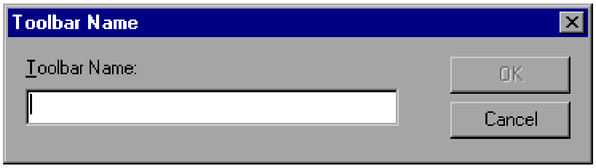
To restore the default buttons for all WinRunner toolbars, click **Reset All**.

To create a user-defined toolbar:

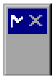
1 Choose **View** > **Customize Toolbars**. The Customize Toolbars dialog box opens and displays the **Commands** tab.

2 Click the **Toolbars** tab.

3 Click **New**. The Toolbar Name dialog box opens.



4 Enter a unique name for the toolbar and click **OK**. The name of the new toolbar is added to the Toolbars list. The new, blank toolbar opens as a *floating* toolbar in the middle of your screen.



5 Drag the toolbar to the location where you want to keep it. If you drag the toolbar to a location within the top or right-hand toolbar area, it becomes a *docked* toolbar (the titlebar is replaced with a toolbar handle).

Tip: You can also double-click the titlebar to dock the toolbar in a default location in the top toolbar area.

---

6 Use the **Commands** tab of the Customize Toolbars dialog box to add toolbar buttons to your new toolbar. For more information, see "Adding or Removing Toolbar Buttons that Perform Menu Commands" on page 832.

To rename a user-defined toolbar:

1 Choose **View** > **Customize Toolbars**. The Customize Toolbars dialog box opens and displays the **Commands** tab.

2 Click the **Toolbars** tab.

3 Select the user-defined toolbar you want to rename.

Note: The **Rename** option is enabled only when a user-defined toolbar is selected.

---

4 Click **Rename**. The Toolbar Name dialog box opens and displays the current name of the selected toolbar.

5 Enter a new name and click **OK**.

To delete a user-defined toolbar:

1 Choose **View** > **Customize Toolbars**. The Customize Toolbars dialog box opens and displays the **Commands** tab.

2 Click the **Toolbars** tab.

3 Select the user-defined toolbar you want to rename.

Note: The **Delete** option is enabled only when a user-defined toolbar is selected.

---

4 Click **Delete**.

5 Click **Yes** to confirm that you want to delete the selected toolbar. The toolbar is deleted from the toolbars list and from the WinRunner window.

To display text labels on the Debug, File, or Test Toolbars:

1 Choose **View** > **Customize Toolbars**. The Customize Toolbars dialog box opens and displays the **Commands** tab.

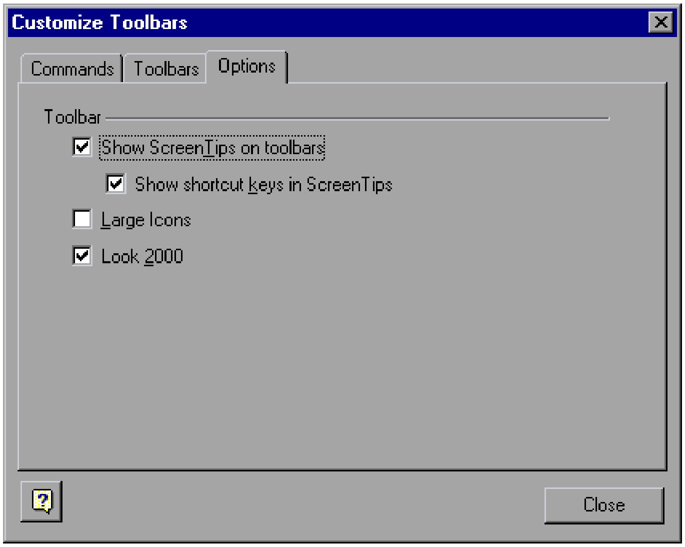
2 Click the **Toolbars** tab.

3 Select the **Debug**, **File**, or **Test** toolbar from the Toolbars list.

4 Select the Show text labels check box.

Setting Toolbar Options

The **Options** tab of the Customize Toolbars dialog box enables you to set options that apply to all toolbars.



Customize Toolbars

☑ Show ScreenTips on toolbars

☑ Show shortcut keys in ScreenTips

□ Large Icons

☑ Look 2000

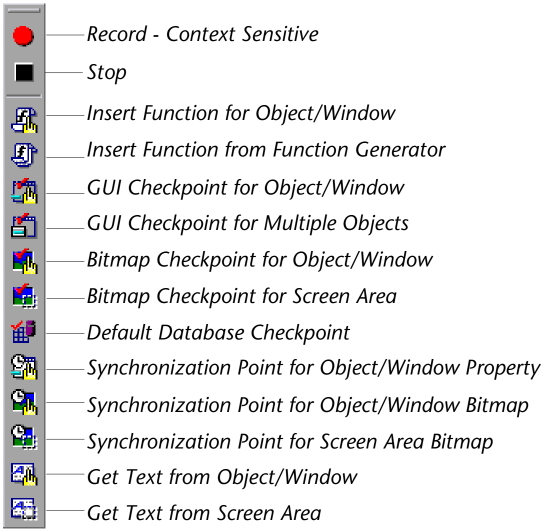
The **Options** tab contains the following options:

Option	Description
Show ScreenTips on toolbars	Shows tips containing the name of the command represented by a toolbar button when you point to the button with the mouse.
Show shortcut keys in ScreenTips	Shows the shortcut key for the command represented by a toolbar button in its screen tip. Enabled only when **Show ScreenTips on toolbars** is selected

Option	Description
Large Icons	Displays all toolbar buttons using large icons.
Look 2000	When selected, displays toolbar handles in the Windows 2000 style with one bar. When cleared, displays toolbar handles with two bars. This option is available only when the **Default** theme is selected in the **Appearance** category of the General Options dialog box.

Customizing the User Toolbar

The User toolbar contains buttons for commands used when creating tests. In its default setting, the User toolbar enables easy access to the following WinRunner commands:


Record - Context Sensitive
Stop
Insert Function for Object/Window
Insert Function from Function Generator
GUI Checkpoint for Object/Window
GUI Checkpoint for Multiple Objects
Bitmap Checkpoint for Object/Window
Bitmap Checkpoint for Screen Area
Default Database Checkpoint
Synchronization Point for Object/Window Property
Synchronization Point for Object/Window Bitmap
Synchronization Point for Screen Area Bitmap
Get Text from Object/Window
Get Text from Screen Area

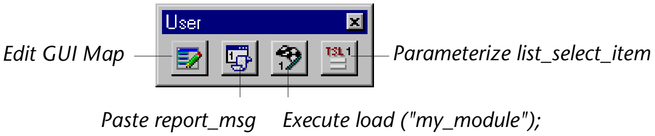
By default, the User toolbar is hidden. To display the User toolbar, choose **View** > **User Toolbar** or select **User Toolbar** in the **Toolbars** tab of the Customize Toolbars dialog box (**View** > **Customize Toolbars**). When the User toolbar is displayed, its default position is docked at the right edge of the WinRunner window.

The User toolbar is a customizable toolbar. You can add or remove buttons to facilitate access to the commands you most frequently use when testing an application. You can use the User toolbar to:

- execute additional WinRunner menu commands. For example, you can add a button to the User toolbar that opens the GUI Map Editor.
- paste TSL statements into your test scripts. For example, you can add a button to the User toolbar that pastes the TSL statement **report_msg** into your test scripts.
- execute TSL statements. For example, you can add a button to the User toolbar that executes the TSL statement:

  ```
  load ("my_module");
  ```

- parameterize TSL statements before pasting them into your test scripts or executing them. For example, you can add a button to the User toolbar that enables you to add parameters to the TSL statement **list_select_item**, and then either paste it into your test script or execute it.



Note: None of the buttons that appear by default in the User toolbar appear in the illustration above.

Adding Buttons to the User Toolbar that Perform Menu Commands

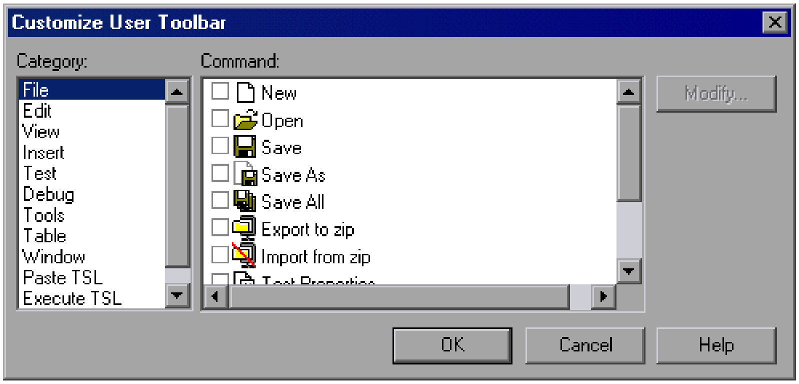
You can add buttons to the User toolbar that perform frequently-used menu commands using the Customize User Toolbar dialog box.

Note: You can also add buttons to the **File** and **Debug** toolbars, and you can create user-defined toolbars. For more information, see “Customizing the File, Debug, and User-Defined Toolbars” on page 832.

To add a menu command to the User toolbar:

1 Choose **View** > **Customize User Toolbar**.

The Customize User Toolbar dialog box opens.



Note that each menu in the menu bar corresponds to a category in the Category pane of the Customize User Toolbar dialog box.

2 In the **Category** pane, select a menu.

3 In the **Command** pane, select the check box next to the menu command.

4 Click **OK** to close the Customize User Toolbar dialog box.

The selected menu command button is added to the User toolbar.

To remove a menu command from the User toolbar:

1 Choose **View** > **Customize User Toolbar** to open the Customize User Toolbar dialog box.

2 In the **Category** pane, select a menu.

3 In the **Command** pane, clear the check box next to the menu command.

4 Click **OK** to close the Customize User Toolbar dialog box.

The selected menu command button is removed from the User toolbar.

Tip: You can also restore the default buttons to the User toolbar using the **Reset** or **Reset All** buttons in the **Toolbars** tab of the Customize Toolbars dialog box. For more information, see "Controlling the Toolbars Display" on page 834.

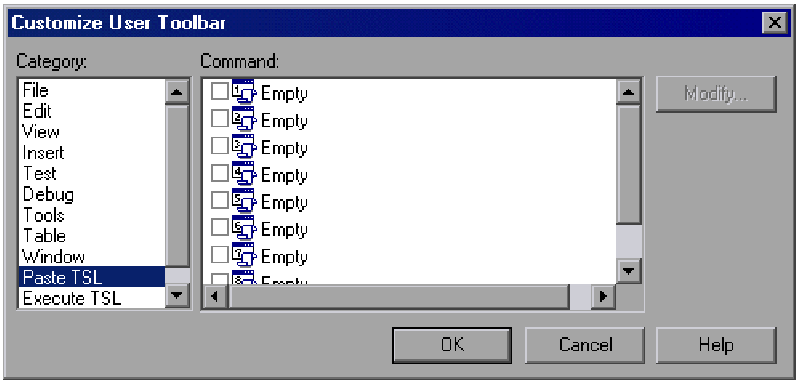
Adding Buttons that Paste TSL Statements

You can add buttons to the User toolbar that paste TSL statements into test scripts. One button can paste a single TSL statement or a group of statements.

To add a button to the User toolbar that pastes TSL statements:

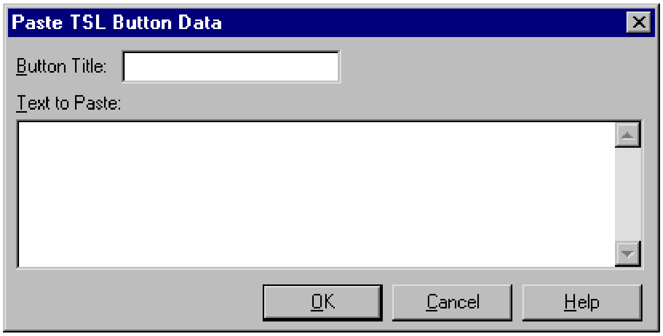
1 Choose **View** > **Customize User Toolbar**. The Customize User Toolbar dialog box opens.

2 In the **Category** pane, select **Paste TSL**.



3 In the **Command** pane, select the check box next to a button, and then select the button.

4 Click **Modify**. The Paste TSL Button Data dialog box opens.



5 In the **Button Title** box, enter a name for the button.

6 In the **Text to Paste** pane, enter the TSL statement(s).

7 Click **OK** to close the Paste TSL Button Data dialog box.

The name of the button is displayed beside the corresponding button in the Command pane.

8 Click **OK** to close the Customize User Toolbar dialog box. The button is added to the User toolbar.

To modify a button on the User toolbar that pastes TSL statements:

1. Choose **View** > **Customize User Toolbar** to open the Customize User Toolbar dialog box.
2. In the **Category** pane, select **Paste TSL**.
3. In the **Command** pane, select the button whose content you want to modify.
4. Click **Modify**. The Paste TSL Button Data dialog box opens.
5. Enter the desired changes in the **Button Title** box and/or the **Text to Paste** pane.
6. Click **OK** to close the Paste TSL Button Data dialog box.
7. Click **OK** to close the Customize User Toolbar dialog box. The button on the User toolbar is modified.

To remove a button from the User toolbar that pastes TSL statements:

1. Choose **View** > **Customize User Toolbar** to open the Customize User Toolbar dialog box.
2. In the **Category** pane, select **Paste TSL**.
3. In the **Command** pane, clear the check box next to the button.
4. Click **OK** to close the Customize User Toolbar dialog box. The button is removed from the User toolbar.

Adding Buttons that Execute TSL Statements

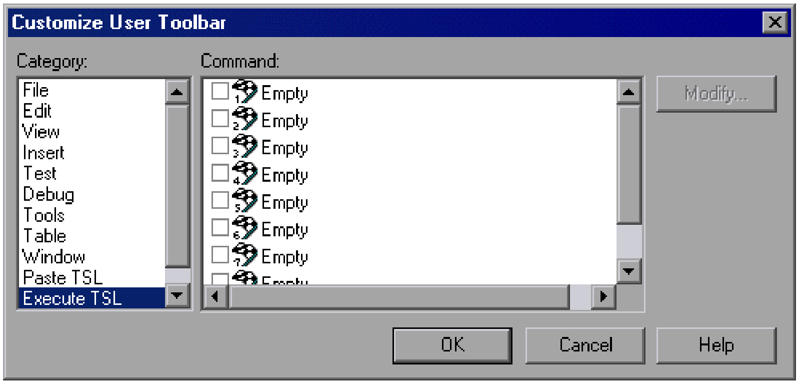
You can add buttons to the User toolbar that execute frequently-used TSL statements.

To add a button to the User toolbar that executes a TSL statement:

1. Choose **View** > **Customize User Toolbar**.

   The Customize User Toolbar dialog box opens.

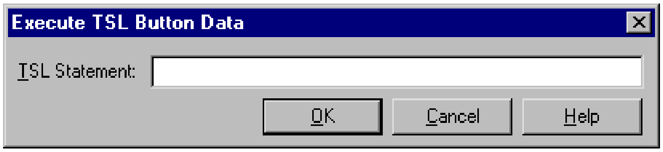
2 In the **Category** pane, select **Execute TSL**.



3 In the **Command** pane, select the check box next to a button, and then select the button.

4 Click **Modify**.

The Execute TSL Button Data dialog box opens.



5 In the **TSL Statement** box, enter the TSL statement.

6 Click **OK** to close the Execute TSL Button Data dialog box.

The TSL statement is displayed beside the corresponding button in the Command pane.

7 Click **OK** to close the Customize User Toolbar dialog box. The button is added to the User toolbar.

To modify a button on the User toolbar that executes a TSL statement:

1 Choose **View** > **Customize User Toolbar** to open the Customize User Toolbar dialog box.

2 In the **Category** pane, select **Execute TSL**.

3 In the **Command** pane, select the button whose content you want to modify.

4 Click **Modify**. The Execute TSL Button Data dialog box opens.

5 Enter the desired changes in the **TSL Statement** box.

6 Click **OK** to close the Execute TSL Button Data dialog box.

7 Click **OK** to close the Customize User Toolbar dialog box. The button on the User toolbar is modified.

To remove a button from the User toolbar that executes a TSL statement:

1 Choose **View** > **Customize User Toolbar** to open the Customize User Toolbar dialog box.

2 In the **Category** pane, select **Execute TSL**.

3 In the **Command** pane, clear the check box next to the button.

4 Click **OK** to close the Customize User Toolbar dialog box. The button is removed from the User toolbar.

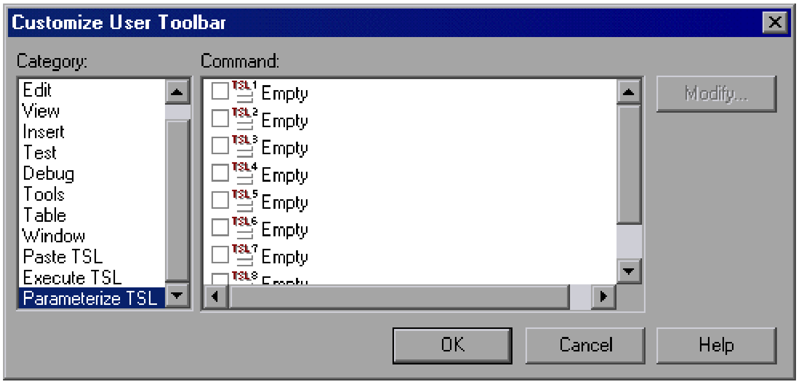
Adding Buttons that Parameterize TSL Statements

You can add buttons to the User toolbar that enable you to easily parameterize frequently-used TSL statements, and then paste them into your test script or execute them.

To add a button to the User toolbar that enables you to parameterize a TSL statement:

1 Choose **View** > **Customize User Toolbar**. The Customize User Toolbar dialog box opens.

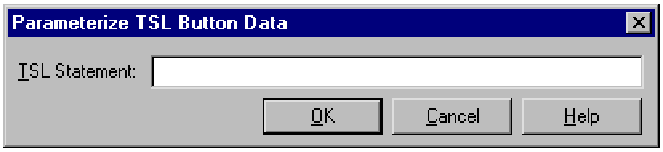
2 In the **Category** pane, select **Parameterize TSL**.



3 In the **Command** pane, select the check box next to a button, and then select the button.

4 Click **Modify**.

The Parameterize TSL Button Data dialog box opens.



5 In the **TSL Statement** box, enter the name of TSL function. You do not need to enter any parameters. For example, enter **list_select_item**.

6 Click **OK** to close the Parameterize TSL Button Data dialog box. The TSL statement is displayed beside the corresponding button in the Command pane.

7 Click **OK** to close the Customize User Toolbar dialog box. The button is added to the User toolbar.

To modify a button on the User toolbar that enables you to parameterize a TSL statement:

1. Choose **View** > **Customize User Toolbar** to open the Customize User Toolbar dialog box.
2. In the **Category** pane, select **Parameterize TSL**.
3. In the **Command** pane, select the button whose content you want to modify.
4. Click **Modify**. The Parameterize TSL Button Data dialog box opens.
5. Enter the desired changes in the **TSL Statement** box.
6. Click **OK** to close the Parameterize TSL Button Data dialog box.
7. Click **OK** to close the Customize User Toolbar dialog box. The button on the User toolbar is modified.

To remove a button from the User toolbar that enables you to parameterize a TSL statement:

1. Choose **View** > **Customize User Toolbar** to open the Customize User Toolbar dialog box.
2. In the **Category** pane, select **Parameterize TSL**.
3. In the **Command** pane, clear the check box next to the button.
4. Click **OK** to close the Customize User Toolbar dialog box. The button is removed from the User toolbar.

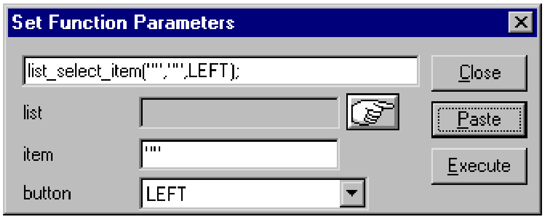
Using the User Toolbar

The User toolbar is hidden by default. You can display it by selecting it from the **View** menu. To execute a command on the User toolbar, click the button that corresponds to the command you want. You can also access the same TSL-based commands that appear on the User toolbar by choosing them on the **Insert** menu.

When the User toolbar is a "floating" toolbar, it remains open when you minimize WinRunner while recording a test. For additional information, see Chapter 11, "Designing Tests."

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.



list_select_item("","",LEFT);

list

item ""

button LEFT

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, click the pointing hand. WinRunner is minimized, a help window opens, and the mouse pointer becomes a pointing hand. Click the list in your application.
- To define a value for the *item* parameter, type it in the corresponding box.
- To define a value for the *button* parameter, select it from the list.

Accessing TSL Statements on the Menu Bar

All TSL statements that you add to the User toolbar can also be accessed via the **Insert** menu.

To choose a TSL statement from a menu:

- To paste a TSL statement, you click **Insert** > **Paste TSL** > [TSL Statement].
- To execute a TSL statement, you click **Insert** > **Execute TSL** > [TSL Statement].
- To parameterize a TSL statement, you click **Insert** > **Parameterize TSL** > [TSL Statement].

Configuring WinRunner Softkeys

Several WinRunner commands can be carried out using softkeys. WinRunner can carry out softkey commands even when the WinRunner window is not the active window on your screen, or when it is minimized.

If the application you are testing uses a softkey combination that is preconfigured for WinRunner, you can redefine the WinRunner softkey combination using WinRunner's Softkey Configuration utility.

Default Settings for WinRunner Softkeys

The following table lists the default softkey configurations and their functions.

Command	Default Softkey Combination	Function
RECORD	F2	Starts test recording. While recording, this softkey toggles between Context Sensitive and Analog modes.
CHECK GUI FOR SINGLE PROPERTY	Alt Right + F12	Checks a single property of a GUI object.
CHECK GUI FOR OBJECT/WINDOW	Ctrl Right + F12	Creates a GUI checkpoint for an object or a window.
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.

Command	Default Softkey Combination	Function
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.
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.
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.

Command	Default Softkey Combination	Function
STEP TO CURSOR	Ctrl Left + F9	Runs a test from the line indicated by the arrow to the line marked by the insertion point.
PAUSE	PAUSE	Stops the test run after all previously interpreted TSL statements have been executed. Execution can be resumed from this point using the Run from Arrow command or the RUN FROM ARROW softkey.
STOP	Ctrl Left + F3	Stops test recording or the test run.
MOVE LOCATOR	Alt Left + F6	Records a move_locator_abs statement with the current position (in pixels) of the screen pointer.

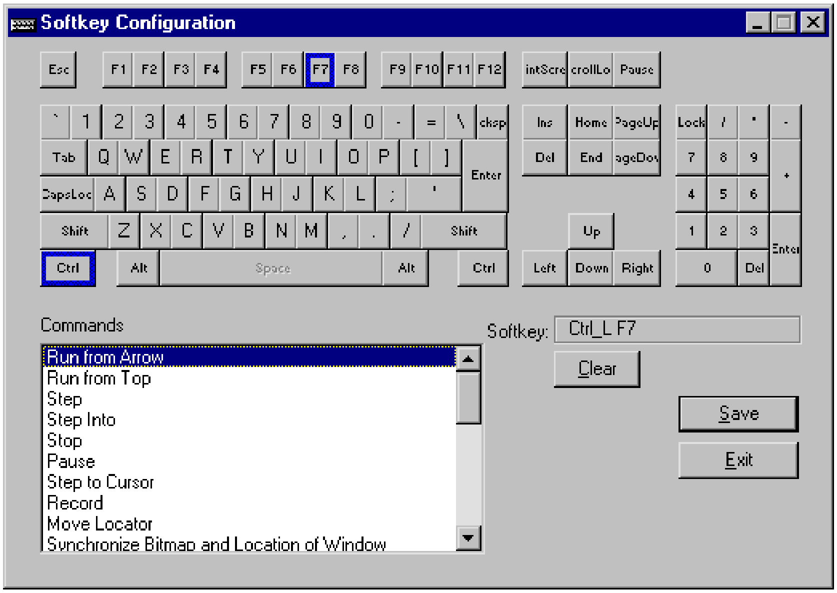
Redefining WinRunner Softkeys

The Softkey Configuration dialog box lists the current softkey assignments and displays an image of a keyboard. To change a softkey setting, 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 softkey commands.



Softkey Configuration

Commands

Run from Arrow
Run from Top
Step
Step Into
Stop
Pause
Step to Cursor
Record
Move Locator
Synchronize Bitmap and Location of Window

Softkey: Ctrl_L F7

Esc F1 F2 F3 F4 F5 F6 F7 F8 F9 F10 F11 F12 intScre crollLo Pause
` 1 2 3 4 5 6 7 8 9 0 - = \ cksp Ins Home PageUp Lock / * -
Tab Q W E R T Y U I O P [ ] Enter Del End ageDov 7 8 9 +
CapsLoc A S D F G H J K L ; ' 4 5 6
Shift Z X C V B N M , . / Shift Up 1 2 3 Enter
Ctrl Alt Space Alt Ctrl Left Down Right 0 Del
Clear
Save
Exit

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.

44

Setting Testing Options from a Test Script

You can control how WinRunner records and runs tests by setting and retrieving testing options from within a test script.

This chapter describes:

About Setting Testing Options from a Test Script

WinRunner testing options affect how you record test scripts and run tests. For example, you can set the speed at which WinRunner executes a test or determine how WinRunner records keyboard input.

You can set and retrieve the values of testing options from within a test script. To set the value of a testing option, use the **setvar** function. To retrieve the current value of a testing option, use the **getvar** function. By using a combination of **setvar** and **getvar** statements in a test script, you can control how WinRunner executes a test. You can use these functions to set and view the testing options for all tests, for a single test, or for part of a single test. You can also use these functions in a startup test script to set environment variables.

Most testing options can also be set using the General Options dialog box. For more information on setting testing options using the General Options dialog box, see Chapter 41, "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
attached_text_search_radius
beep
capture_bitmap
cs_run_delay
cs_fail
delay_msec
drop_sync_timeout
email_service
enum_descendent_toplevel
fontgrp
item_number_seq
List_item_separator
Listview_item_separator
min_diff
mismatch_break
rec_item_name
rec_owner_drawn
searchpath
silent_mode
single_prop_check_fail
speed
sync_fail_beep
synchronization_timeout
tempdir
timeout_msec
Treeview_path_separator

For example, if you execute the following **setvar** statement:

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

WinRunner disables the *mismatch_break* testing option. The setting remains in effect during the testing session until it is changed again, either with another **setvar** statement or from the corresponding **Break when verification fails** check box in the **Run** > **Settings** category of the General Options dialog box.

Using the **setvar** function changes a testing option globally, and this change is reflected in the General Options dialog box. However, you can also use the **setvar** function to set testing options for a specific test, or even for part of a specific test.

To use the **setvar** function to change a variable only for the current test, without overwriting its global value, save the original value of the variable separately and restore it later in the test.

Note: Some testing options are set by WinRunner and cannot be changed through either **setvar** or the General Options dialog box. For example, the value of the testname option is always the name of the current test. You can use **getvar** to retrieve this read-only value. For more information, see “Retrieving Testing Options with getvar” on page 858.

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) ;
```

Retrieving Testing Options with getvar

You use the **getvar** function to retrieve the current value of a testing option. The **getvar** function is a read-only function, and does not enable you to alter the value of the retrieved testing option. (To change the value of a testing option in a test script, use the **setvar** function, described above.) The syntax of this statement is:

user_variable = **getvar (**"*testing_option*"**);**

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

attached_text_area
attached_text_search_radius
batch
beep
capture_bitmap
cs_fail
cs_run_delay
curr_dir
delay_msec
drop_sync_timeout
email_service
enum_descendent_toplevel
exp
fontgrp
item_number_seq
key_editing
line_no
List_item_separator
Listview_item_separator
min_diff
mismatch_break
rec_item_name
rec_owner_drawn
result
runmode
searchpath
shared_checklist_dir
single_prop_check_fail
silent_mode
speed
sync_fail_beep
synchronization_timeout
td_log_dirname
td_connection
td_cycle_name
td_database_name
td_server_name
td_user_name
tempdir
testname
timeout_msec
Treeview_path_separator

For example:

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

assigns the current value of the run speed to the user-defined variable currspeed.

Controlling the Test Run with setvar and getvar

You can use **getvar** and **setvar** together to control a test run without changing global settings. In the following test script fragment, WinRunner checks the bitmap Img1. The **getvar** function retrieves the values of the *timeout_msec* and *delay_msec* testing options, and **setvar** assigns their values for this **win_check_bitmap** statement. After the window is checked, **setvar** restores the values of the testing options.

```
t = getvar ("timeout_msec");
d = getvar ("delay_msec");
setvar ("timeout_msec", 30000);
setvar ("delay_msec", 3000);
win_check_bitmap ("calculator", Img1, 2, 261,269,93,42);
setvar ("timeout_msec", t);
setvar ("delay_msec", d);
```

Note: You can use the **setvar** and **getvar** functions in a startup test script to set environment variables for a specific WinRunner session. For more information, see Chapter 46, "Initializing Special Configurations."

Test Script Testing Options

This section describes the WinRunner testing options that can be used with the **setvar** and **getvar** functions from within a test script. If you can also use set or view the corresponding option from a dialog box, it is indicated below.

attached_text_area

This option specifies the location on a GUI object from which WinRunner searches for its attached text.

Possible values:

Value	**Location on the GUI Object**
Default	Top-left corner of regular (English-style) windows; Top-right corner of windows with RTL-style (WS_EX_BIDI_CAPTION) windows.
Top-Left	Top-left corner.
Top	Midpoint of two top corners.
Top-Right	Top-right corner.
Right	Midpoint of two right corners.
Bottom-Right	Bottom-right corner.
Bottom	Midpoint of two bottom corners.
Bottom-Left	Bottom-left corner.
Left	Midpoint of two left corners.

Note: All of the above possible values are text strings.

You can use this option with the **setvar** and **getvar** functions.

You can also set this option using the **Attached Text - Preferred search area** box in the **Record** category of the General Options dialog box as described in “Setting Recording Options” on page 778.

Notes: When you run a test, you must use the same values for the attached text options that you used when you recorded the test. Otherwise, WinRunner may not identify the GUI object.

In previous versions of WinRunner, you could not set the preferred search area: WinRunner searched for attached text based on what is now the Default setting for the preferred search area. If backward compatibility is important, choose the Default setting.

attached_text_search_radius

This option specifies the radius from the specified location on a GUI object that WinRunner searches for the static text object that is its attached text.

Possible values: 3 - 300 (pixels)

You can use this option with the **setvar** and **getvar** functions.

You can also set this option using the **Attached Text - Search radius** box in the **Record** category of the General Options dialog box as described in "Setting Recording Options" on page 778.

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 35, "Running Batch Tests."

For example, if a **set_window** statement is missing from a test script, WinRunner cannot find the specified window. If this option is on and the test is run in batch mode, WinRunner reports an error in the Test Results window and proceeds to run the next statement in the test script. If this option is off and the test is not run in batch mode, WinRunner pauses the test and opens the Run wizard to enable the user to locate the window.

You can use this option with the **getvar** function.

Possible values: on, off (text strings)

You can also set this option using the **Run in batch mode** check box in the **Run** category of the General Options dialog box as described in "Setting Test Run Options" on page 793.

Note that you can also set this option using the corresponding *-batch* command line option, described in Chapter 36, "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 873.

beep

This option determines whether WinRunner beeps when checking any window during a test run.

You can use this option with the **setvar** and **getvar** functions.

Possible values: on, off (text strings)

You can also set this option using the corresponding **Beep when checking a window** check box in the **Run** > **Settings** category of the General Options dialog box as described in "Setting Run Setting Options" on page 797.

Note that you can also set this option using the corresponding *-beep* command line option, described in Chapter 36, "Running Tests from the Command Line."

capture_bitmap

This option determines whether WinRunner captures a bitmap whenever a checkpoint fails. When this option is on, WinRunner uses the settings from the **Run** > **Settings** category of the General Options dialog box to determine the captured area for the bitmaps.

You can use this option with the **setvar** and **getvar** functions.

Possible values: on, off (text strings)

You can also set this option using the **Capture bitmap on verification failure** check box in the **Run** > **Settings** category of the General Options dialog box, as described in "Setting Run Setting Options" on page 797.

Note that you can also set this option using the corresponding -*capture_bitmap* command line option, described in Chapter 36, "Running Tests from the Command Line."

cs_fail

This option determines whether WinRunner fails a test when Context Sensitive errors occur. A Context Sensitive error is the failure of a Context Sensitive statement during a test. Context Sensitive errors are often due to WinRunner's failure to identify a GUI object.

For example, a Context Sensitive error will occur if you run a test containing a **set_window** statement with the name of a non-existent window. Context Sensitive errors can also occur when window names are ambiguous. For information about Context Sensitive functions, refer to the *TSL Reference*.

You can use this option with the **setvar** and **getvar** functions.

Possible values: 1,0

You can also set this option using the corresponding **Fail test when Context Sensitive errors occur** check box in the **Run** > **Settings** category of the General Options dialog box as described in "Setting Run Setting Options" on page 797.

You can also set this option using the corresponding -*cs_fail* command line option, described in Chapter 36, "Running Tests from the Command Line."

cs_run_delay

This option sets the time (in milliseconds) that WinRunner waits between executing Context Sensitive statements when running a test.

You can use this option with the **setvar** and **getvar** functions.

Possible values: numbers 0 and higher

You can also set this option using the corresponding **Delay between execution of CS statements** box in the **Run** > **Synchronization** category of the General Options dialog box as described in "Setting Run Synchronization Options" on page 802.

Note that you can also set this option using the corresponding *-cs_run_delay* command line option, described in Chapter 36, "Running Tests from the Command Line."

curr_dir

This option displays the current working folder for the test.

You can use this option with the **getvar** function.

You can also view the location of the current working folder for the test from the corresponding **Current folder** box in the **Current Test** tab of the Test Properties dialog box, described in "Reviewing Current Test Settings" on page 757.

delay_msec

This option sets the sampling interval (in seconds) used to determine that a window is stable before capturing it for a Context Sensitive checkpoint or synchronization point. To be declared stable, a window must not change between two consecutive samplings. This sampling continues until the window is stable or the timeout (as set with the *timeout_msec* testing option) is reached. (Formerly *delay*, which was measured in seconds.)

For example, when the delay is two seconds and the timeout is ten seconds, WinRunner checks the window in the application under test every two seconds until two consecutive checks produce the same results or until ten seconds have elapsed. Setting the value to 0 disables all bitmap checking.

You can use this option with the **setvar** and **getvar** functions.

Possible values: numbers 0 and higher

Note: This option is accurate to within 20-30 milliseconds.

You can also set this option using the corresponding **Delay for window synchronization** option in the **Run** > **Synchronization** category of the General Options dialog box as described in “Setting Run Synchronization Options” on page 802.

Note that you can also set this option using the corresponding *-delay_msec* command line option, described in Chapter 36, “Running Tests from the Command Line.”

drop_sync_timeout

determines whether WinRunner minimizes the synchronization timeout (as defined in the **timeout_msec** option) after the first synchronization failure.

Possible values: on, off (text strings)

You can use this option with the **getvar** and **setvar** functions.

You can also set this option using the corresponding **Drop synchronization timeout if failed** check box in the **Run** > **Synchronization** category of the General Options dialog box as described in “Setting Run Synchronization Options” on page 802.

email_service

This option determines whether WinRunner activates the e-mail sending options including the e-mail notifications for checkpoint failures, test failures, and test completed reports as well as any **email_send_msg** statements in the test.

Possible values: on, off (text strings)

You can use this option with the **getvar** and **setvar** functions.

You can also set this option using the corresponding Activate e-mail service check box in the **Notifications** > **E-mail** category of the General Options dialog box as described in "Setting E-mail Notification Options" on page 810.

Note that you can also set this option using the corresponding **-email_service** command line option, described in Chapter 36, "Running Tests from the Command Line."

enum_descendent_toplevel

This option determines whether WinRunner records controls (objects) of a child object whose parent is an object but not a window and identifies these controls when running a test.

Possible values: 1,0

You can use this option with the **getvar** and **setvar** functions.

You can also set this option using the corresponding **Consider child windows** check box in the **Record** category of the General Options dialog box as described in "Setting Recording Options" on page 778.

exp

This option displays the full path of the expected results folder associated with the current test run.

You can use this option with the **getvar** function.

You can also view the full path of the expected results folder from the corresponding **Expected results folder** box in the **Current Test** tab of the Test Properties dialog box as described in "Reviewing Current Test Settings" on page 757.

Note that you can also set this option using the corresponding *-exp* command line option, described in Chapter 36, "Running Tests from the Command Line."

fontgrp

To be able to use Image Text Recognition (instead of the default Text Recognition), (described in "Setting Text Recognition Options" on page 790), 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 395.

You can use this option with the **setvar** and **getvar** functions.

Possible values: any text string

You can also set this option using the corresponding **Font group** box in the **Record** > **Text Recognition** category of the General Options dialog box as described in "Setting Text Recognition Options" on page 790.

Note that you can also set this option using the corresponding *-fontgrp* command line option, described in Chapter 36, "Running Tests from the Command Line."

item_number_seq

This option defines the string recorded in the test script to indicate that a List, ListView, or TreeView item is specified by its index number.

You can use this option with the **setvar** and **getvar** functions.

Possible values: any text string

You can also set this option using the corresponding **String indicating that what follows is a number** box in the **Record** > **Script Format** category of the General Options dialog box as described in "Setting Script Format Options" on page 786.

key_editing

This option determines whether WinRunner generates more concise **type**, **win_type**, and **obj_type** statements in a test script.

When this option is on, WinRunner generates more concise **type**, **win_type**, and **obj_type** statements that represent only the net result of pressing and releasing input keys. This makes your test script easier to read.

For example:

```
obj_type (object, "A");
```

When this option is disabled, WinRunner records the pressing and releasing of each key. For example:

```
obj_type (object, "<kShift_L>-a-a+<kShift_L>+");
```

Disable this option if the exact order of keystrokes is important for your test.

For more information on this subject, see the **type** function in the *TSL Reference*.

You can use this option with the **setvar** and **getvar** functions.

Possible values: on, off (text strings)

You can also set this option using the corresponding **Generate concise, more readable type statements** check box in the **Record** > **Script Format** category of the General Options dialog box as described in "Setting Script Format Options" on page 786.

line_no

This option displays the line number of the current location of the execution arrow in the test script.

You can use this option with the **getvar** function.

You can also view the current line number in the test script from the corresponding **Current line number** box in the **Current Test** tab of the Test Properties dialog box, described in "Reviewing Current Test Settings" on page 757.

List_item_separator

This option defines the string recorded in the test script to separate items in a list box or a combo box.

You can use this option with the **setvar** and **getvar** functions.

Possible values: any text string

You can also set this option using the corresponding **String for separating ListBox or ComboBox items** box in the **Record** > **Script Format** category of the General Options dialog box as described in "Setting Script Format Options" on page 786.

Listview_item_separator

This option defines the string recorded in the test script to separate items in a ListView or a TreeView.

You can use this option with the **setvar** and **getvar** functions.

Possible values: any text string

You can also set this option using the corresponding **String for separating ListView or TreeView items** box in the **Record** > **Script Format** category of the General Options dialog box as described in "Setting Script Format Options" on page 786.

min_diff

This option defines the number of pixels that constitute the threshold for bitmap mismatch. When this value is set to 0, a single pixel mismatch constitutes a bitmap mismatch.

You can use this option with the **setvar** and **getvar** functions.

Possible values: numbers 0 and higher

You can also set this option using the corresponding **Threshold for difference between bitmaps** box in the **Run** > **Settings** category of the General Options dialog box as described in "Setting Run Setting Options" on page 797.

Note that you can also set this option using the corresponding *-min_diff* command line option, described in Chapter 36, "Running Tests from the Command Line."

mismatch_break

This option determines whether WinRunner pauses the test run and displays a message whenever verification fails or whenever any message is generated as a result of a context sensitive statement during a test that is run in Verify mode. This option should be used only when working interactively.

For example, if a **set_window** statement is missing from a test script, WinRunner cannot find the specified window. If this option is on, WinRunner pauses the test and opens the Run wizard to enable the user to locate the window. If this option is off, WinRunner reports an error in the Test Results window and proceeds to run the next statement in the test script.

You can use this option with the **setvar** and **getvar** functions.

Possible values: on, off (text strings)

You can also set this option using the corresponding **Break when verification fails** check box in the **Run** > **Settings** category of the General Options dialog box as described in "Setting Run Setting Options" on page 797.

Note that you can also set this option using the corresponding *-mismatch_break* command line option, described in Chapter 36, "Running Tests from the Command Line."

rec_item_name

This option determines whether WinRunner records non-unique ListBox and ComboBox items by name or by index.

You can use this option with the **setvar** and **getvar** functions.

Possible values: 1,0

You can also set this option using the corresponding **Record non-unique list items by name** check box in the **Record** category of the General Options dialog box as described in "Setting Recording Options" on page 778.

Note that you can also set this option using the corresponding *-rec_item_name* command line option, described in Chapter 36, “Running Tests from the Command Line.”

rec_owner_drawn

Since WinRunner cannot identify the class of owner-drawn buttons, it automatically maps them to the general “object” class. This option enables you to map all owner-drawn buttons to a standard button class (push_button, radio_button, or check_button).

You can use this option with the **setvar** and **getvar** functions.

Possible Values: object, push_button, radio_button, check_button (text strings)

You can also set this option using the corresponding **Record owner-drawn buttons as** box in the **Record** category of the General Options dialog box as described in “Setting Recording Options” on page 778.

result

This option displays the full path of the verification results folder associated with the current test run.

You can use this option with the **getvar** function.

You can also view the full path of the verification results folder from the corresponding **Verification results folder** box in the **Current Test** tab of the Test Properties dialog box as described in “Reviewing Current Test Settings” on page 757.

runmode

This option displays the current run mode.

You can use this option with the **getvar** function.

Possible values: verify, debug, update (text strings)

You can also view the current run mode from the corresponding **Run mode** box in the **Current Test** tab of the Test Properties dialog box, described in “Reviewing Current Test Settings” on page 757.

searchpath

This option sets the path(s) in which WinRunner searches for called tests. If you define search paths, you do not need to designate the full path of a test in a call statement. You can set multiple search paths in a single statement by leaving a space between each path. To set multiple search paths for long file names, surround each path with angle brackets < >. WinRunner searches for a called test in the order in which multiple paths appear in the **getvar** or **setvar** statement.

You can use this option with the **setvar** and **getvar** functions.

You can also set this option using the corresponding **Search path for called tests** box in the **Folders** category of the General Options dialog box as described in "Setting Folder Options" on page 775.

Note that you can also set this option using the corresponding *-search_path* command line option, described in Chapter 36, "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 190. For more information on shared database checklists, see "Saving a Database Checklist in a Shared Folder" on page 350. Note that if you designate a new folder, you must restart WinRunner in order for the change to take effect.

You can use this option with the **getvar** function.

You can also view the location of the folder in which WinRunner stores shared checklists from the corresponding **Shared checklists** box in the **Folders** category of the General Options dialog box as described in "Setting Folder Options" on page 775.

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 48, "Managing the Testing Process."

You can use this option with the **setvar** and **getvar** functions.

Possible values: on, off (text strings)

Note: When you run tests in batch mode, you automatically run them in silent mode. For information running tests in batch mode, see Chapter 35, "Running Batch Tests."

---

single_prop_check_fail

This option fails a test run when **_check_info** statements fail. It also writes an event to the Test Results window for these statements. (You can create **_check_info** statements using the **Insert** > **GUI Checkpoint** > **For Single Property** command.)

You can use this option with the **setvar** and **getvar** functions.

Possible values: 1,0

For information about the **check_info** functions, refer to the *TSL Reference*.

You can also set this option using the corresponding **Fail test when single property check fails** option in the **Run** > **Settings** category of the General Options dialog box as described in "Setting Run Setting Options" on page 797.

Note that you can also set this option using the corresponding -*single_prop_check_fail* command line option, described in Chapter 36, "Running Tests from the Command Line."

speed

This option sets the default run *s*peed for tests run in Analog mode.

Possible values: normal, fast (text strings)

Setting the option to **normal** runs the test at the speed at which it was recorded.

Setting the option to **fast** runs the test as fast as the application can receive input.

You can use this option with the **setvar** and **getvar** functions.

You can also set this option using the corresponding **Run speed for Analog mode** option in the **Run** category of the General Options dialog box as described in "Setting Test Run Options" on page 793.

Note that you can also set this option using the corresponding -*speed* command line option, described in Chapter 36, "Running Tests from the Command Line."

sync_fail_beep

This option determines whether WinRunner beeps when synchronization fails.

You can use this option with the **setvar** and **getvar** functions.

Possible values: 1,0

You can also set this option using the corresponding **Beep when synchronization fails** check box in the **Run** > **Synchronization** category of the General Options dialog box as described in "Setting Run Synchronization Options" on page 802.

Note: This option is useful primarily for debugging test scripts.

Note: If synchronization often fails during your test runs, consider increasing the value of the *synchronization_timeout* testing option (described below) or the corresponding **Timeout for waiting for synchronization message** option in the **Run** > **Synchronization** category of the General Options dialog box.

synchronization_timeout

This option sets the timeout (in milliseconds) that WinRunner waits before validating that keyboard or mouse input was entered correctly during a test run.

You can use this option with the **setvar** and **getvar** functions.

Possible values: numbers 0 and higher

You can also set this option using the corresponding **Timeout for waiting for synchronization message** box in the **Run** > **Synchronization** category of the General Options dialog box as described in "Setting Run Synchronization Options" on page 802.

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.

Possible values: on, off (text strings)

You can connect to TestDirector from the TestDirector Connection dialog box or using the *-td_connection* command line option. For more information about connecting to TestDirector, see Chapter 48, "Managing the Testing Process."

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.

You can set this option using the Run Tests dialog box when you run a test set from WinRunner while connected to TestDirector. For more information, see "Running Tests in a Test Set" on page 930. You can 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 36, "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.

You can set this option using the **Project** option in the **TestDirector Connection** dialog box, which you can open by choosing **Tools** > **TestDirector Connection**. For more information, see Chapter 48, "Managing the Testing Process."

Note that you can also set this option using the corresponding *-td_database_name* command line option, described in Chapter 36, "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.

You can set this option using the **Server** box in the TestDirector Connection dialog box, which you can open by choosing **Tools** > **TestDirector Connection**. For more information, see Chapter 48, "Managing the Testing Process."

Note that you can also set this option using the corresponding *-td_server_name* command line option, described in Chapter 36, "Running Tests from the Command Line."

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.

Note that you can also set this option using the corresponding *-td_user_name* command line option, described in Chapter 36, "Running Tests from the Command Line."

You can set this option using the **User name** box in the TestDirector Connection dialog box, which you can open by choosing **Tools** > **TestDirector Connection**. For more information, see Chapter 48, "Managing the Testing Process."

tempdir

This option designates the folder containing temporary files. Note that if you designate a new folder, you must restart WinRunner in order for the change to take effect.

You can use this option with the **setvar** and **getvar** functions.

You can also set this option using the corresponding **Temporary files** box in the **Folders** category of the General Options dialog box as described in Chapter 41, "Setting Folder Options."

testname

This option displays the full path of the current test.

You can use this option with the **getvar** function.

You can also view the location and the test name of the current test in the **General** tab of the Test Properties dialog box as described in "Documenting General Test Information" on page 749.

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) milliseconds.

You can use this option with the **setvar** and **getvar** functions.

Possible values: numbers 0 and higher

Note: This option is accurate to within 20-30 milliseconds.

You can also set this option using the corresponding **Timeout for checkpoints and CS statements** box in the **Run** > **Settings** category of the General Options dialog box as described in "Setting Run Setting Options" on page 797.

Note that you can also set this option using the corresponding -*timeout_msec* command line option, described in Chapter 36, "Running Tests from the Command Line."

Treeview_path_separator

This option defines the string recorded in the test script to separate items in a tree view path.

Possible values: any text string

You can use this option with the **getvar** and **setvar** functions.

You can also set this option using the corresponding **String for parsing a TreeView path** box in the **Record** > **Script Format** category of the General Options dialog box as described in "Setting Script Format Options" on page 786.

45

Customizing the Function Generator

You can customize the Function Generator to include the user-defined functions that you most frequently use in your tests scripts. This makes programming tests easier and reduces the potential for errors.

This chapter describes:

- About Customizing the Function Generator
- Adding a Category to the Function Generator
- Adding a Function to the Function Generator
- Associating a Function with a Category
- Adding a Subcategory to a Category
- Setting a Default Function for a Category

About Customizing the Function Generator

You can modify the Function Generator to include the user-defined functions that you use most frequently. This enables you to quickly generate your favorite functions and insert them directly into your test scripts. You can also create custom categories in the Function Generator in which you can organize your user-defined functions. For example, you can create a category named **my_button**, which contains all the functions specific to the **my_button** custom class. You can also set the default function for the new category, or modify the default function for any standard category.

To add a new category with its associated functions to the Function Generator:

1. Add a new category to the Function Generator.
2. Add new functions to the Function Generator.
3. Associate the new functions with the new category.
4. Set the default function for the new category.
5. Add a subcategory for the new category (optional).

You can find all the functions required to customize the Function Generator in the "function table" category of the Function Generator. By inserting these functions in a startup test, you ensure that WinRunner is invoked with the correct configuration.

Adding a Category to the Function Generator

You use the **generator_add_category** TSL function to add a new category to the Function Generator. This function has the following syntax:

generator_add_category (*category_name* **);**

where *category_name* is the name of the category that you want to add to the Function Generator.

In the following example, the **generator_add_category** function adds a category called "my_button" to the Function Generator:

```
generator_add_category ("my_button");
```

Note: If you want to display the default function for category when you select an object using the **Insert** > **Function** > **For Object/Window** command, 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 **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 892.

You use the **generator_add_function** TSL function to add a user-defined function to the Function Generator.

To add a function to the Function Generator:

1. Open the Function Generator. (Choose **Insert** > **Function** > **From Function Generator**, click the **Insert Function from Function Generator** button on the User toolbar, or press the INSERT FUNCTION FROM FUNCTION GENERATOR softkey.)
2. In the **Category** box, click **function table**.
3. In the **Function Name** box, click **generator_add_function**.
4. Click **Args**. The Function Generator expands.
5. In the Function Generator, define the *function_name*, *description*, and *arg_number* arguments:
   - In the *function_name* box, type the name of the new function between the quotes. Note that you can include spaces and upper-case letters in the function name.
   - In the *description* box, enter the description of the function between the quotes. Note that it does not have to be a valid string expression and it must not exceed 180 characters.
   - In the *arg_number* box, you must choose 1. To define additional arguments (up to eight arguments for each new function), you must manually modify the generated **generator_add_function** statement once it is added to your test script.
6. For the function's first argument, define the following arguments: *arg_name*, *arg_type*, and *default_value* (if relevant):
   - In the *arg_name* box, type the name of the argument within the quotation marks. Note that you can include spaces and upper-case letters in the argument name.
   - In the *arg_type* box, select "***browse()***", "***point_object***", "***point_window***", "***select_list (01)***", or "***type_edit***", to choose how the user will fill in the argument's value in the Function Generator, as described in "Defining Function Arguments" on page 885.
   - 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. , , , , or “”

  “browse()”: The value of the argument is evaluated by pointing to a file in a browse file dialog box. Use *browse* when the argument is a file. To select a file with specific file extensions only, specify a list of default extension(s). Items in the list should be separated by a space or tab. Once a new function is defined, the *browse* argument is defined in the Function Generator by using a Browse button.

  “point_object”: The value of the argument is evaluated by pointing to a GUI object (other than a window). Use *point_object* when the argument is the logical name of an object. Once a new function is defined, the *point_object* argument is defined in the Function Generator by using a pointing hand.

  “point_window”: The value of the argument is evaluated by pointing to a window. Use *point_window* when the argument is the logical name of a window. Once a new function is defined, the *point_window* argument is defined in the Function Generator by using a pointing hand.

  “select_list (01)”: The value of the argument is selected from a list. Use *select_list* when there is a limited number of argument values, and you can supply all the values. Once a new function is defined, the *select_list* argument is defined in the Function Generator by using a combo box.

  “type_edit”: The value of the argument is typed in. Use *type_edit* when you cannot supply the full range of argument values. Once a new function is defined, the *type_edit* argument is defined in the Function Generator by typing into an edit field.

- *default_value* provides the argument's default value. You may assign default values to **select_list** and **type_edit** arguments. The default value you specify for a **select_list** argument must be one of the values included in the list. You cannot assign default values to **point_window** and **point_object** arguments.

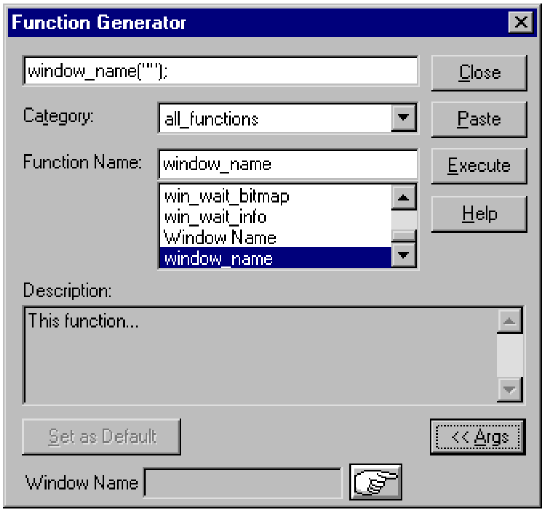
The following are examples of argument definitions that you can include in **generator_add_function** statements. The examples include the syntax of the argument definitions, their representations in the Function Generator, and a brief description of each definition.

Example 1

```
generator_add_function ("window_name","This function...",1,
    "Window Name","point_window","");
```

The *function_name* is window_name. The *description* is “This function...”. The *arg_number* is 1. The *arg_name* is Window Name. The *arg_type* is point_window. There is no *default_value*: since the argument is selected by pointing to a window, this argument is an empty string.

When you select the **window_name** function in the Function Generator and click the Args button, the Function Generator appears as follows:



Function Generator

window_name("");

Category: all_functions

Function Name: window_name

win_wait_bitmap
win_wait_info
Window Name
window_name

Description:

This function...

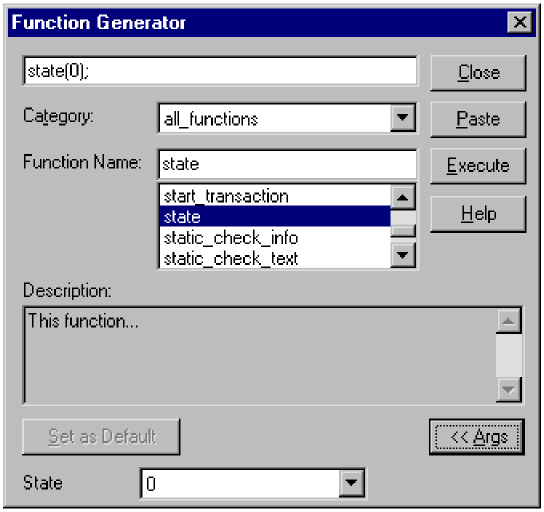
Window Name

Example 2

generator_add_function("state","This function...",1,"State","select_list (0 1)",0);

The *function_name* is state. The *description* is "This function...". The *arg_number* is 1. The *arg_name* is State. The *arg_type* is select_list. The *default_value* is 0.

When you select the **state** function in the Function Generator and click the Args button, the Function Generator appears as follows:



state(0);

Category: all_functions

Function Name: state

Description: This function...

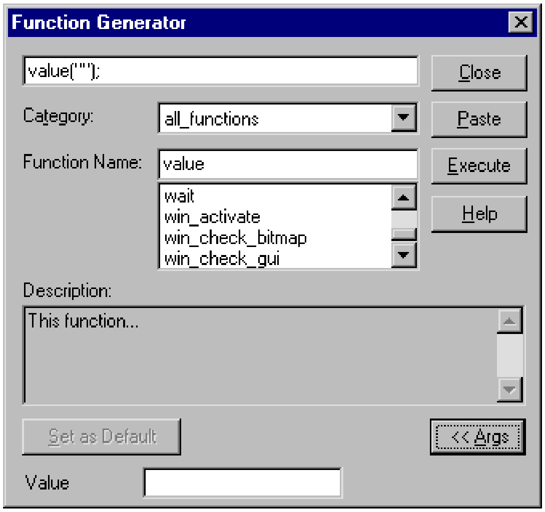
State 0

Example 3

```
generator_add_function("value","This function...",1,"Value","type_edit","");
```

The *function_name* is value. The *description* is "This function...". The *arg_number* is 1. The *arg_name* is Value. The *arg_type* is type_edit. There is no *default_value*.

When you select the **value** function in the Function Generator and click the Args button, the Function Generator appears as follows:



Function Generator

value("");

Category: all_functions

Function Name: value

Description:

This function...

Value

Defining Property Arguments

You can define a function with an argument that uses a Context Sensitive property, such as the label on a pushbutton or the width of a checkbox. In such a case, you cannot define a single default value for the argument. However, you can use the **attr_val** function to determine the value of a property for the selected window or GUI object. You include the **attr_val** function in a call to the **generator_add_function** function.

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

attr_val (*object_name, "property"* **);**

- *object_name* defines the window or GUI object whose property is returned. It must be identical to the *arg_name* defined in a previous argument of the **generator_add_function** function.
- *property* can be any property used in Context Sensitive testing, such as height, width, label, or value. You can also specify platform-specific properties such as MSW_class and MSW_id.

  You can either define a specific property, or specify a parameter that was defined in a previous argument of the same call to the function, **generator_add_function**. For an illustration, see example 2, below.

Example 1

In this example, a function called "check_my_button_label" is added to the Function Generator. This function checks the label of a button.

```
generator_add_function("check_my_button_label", "This function checks the
label of a button.", 2,
    "button_name", "point_object"," ",
    "label", "type_edit", "attr_val(button_name, \"label\")");
```

The "check_my_button_label" function has two arguments. The first is the name of the button. Its selection method is *point_object* and it therefore has no default value. The second argument is the label property of the button specified, and is a *type_edit* argument. The **attr_val** function returns the label property of the selected GUI object as the default value for the property.

Example 2

The following example adds a function called “check_my_property” to the Function Generator. This function checks the *class*, *label*, or *active* property of an object. The property whose value is returned as the default depends on which property is selected from the list.

```
generator_add_function ("check_my_property","This function checks an object’s
property.",3,
    "object_name", "point_object", " ",
    "property", "select_list(\"class\"\"label\"\"active\")", "\"class\"",
    "value:", "type_edit", "attr_val(object_name, property)");
```

The first three arguments in **generator_add_function** define the following:

- the name of the new function (check_my_property).
- the description appearing in the Description field of the Function Generator. This function checks an object’s property.
- the number of arguments (3).

The first argument of “check_my_property” determines the object whose property is to be checked. The first parameter of this argument is the object name. Its type is *point_object*. Consequently, as the null value for the third parameter of the argument indicates, it has no default value.

The second argument is the property to be checked. Its type is *select_list*. The items in the list appear in parentheses, separated by field separators and in quotation marks. The default value is the class property.

The third argument, value, is a *type_edit* argument. It calls the **attr_val** function. This function returns, for the object defined as the function’s first argument, the property that is defined as the second argument (class, label or active).

Associating a Function with a Category

Any function that you add to the Function Generator should be associated with an existing category. You make this association using the **generator_add_function_to_category** TSL function. Both the function and the category must already exist.

This function has the following syntax:

generator_add_function_to_category (*category_name*, *function_name* **);**

- *category_name* is the name of a category in the Function Generator. It can be either a standard category, or a custom category that you defined using the **generator_add_category** function.
- *function_name* is the name of a custom function. You must have already added the function to the Function Generator using the function, **generator_add_function**.

To associate a function with a category:

1. Open the Function Generator. (Choose **Insert** > **Function** > **From Function Generator**, click the **Insert Function from Function Generator** button on the User toolbar, or press the INSERT FUNCTION FROM FUNCTION GENERATOR softkey.)
2. In the **Category** box, click **function table**.
3. In the **Function Name** box, click **generator_add_function_to_category**.
4. Click **Args**. The Function Generator expands.
5. In the **Category Name** box, enter the category name as it already appears in the Function Generator.
6. In the **Function Name** box, enter the function name as it already appears in the Function Generator.
7. Click **Paste** to paste the TSL statement into your test script.
8. Click **Close** to close the Function Generator.

A **generator_add_function_to_category** statement is inserted into your test script. In the following example, the "check_my_button_label" function is associated with the "my_button" category. This example assumes that you have already added the "my_button" category and the "check_my_button_label" function to the Function Generator.

```
generator_add_function_to_category ("my_button", "check_my_button_label");
```

Note: You must run the test script in order to associate a function with a category.

Adding a Subcategory to a Category

You use the **generator_add_subcategory** TSL function to make one category a subcategory of another category. Both categories must already exist. The **generator_add_subcategory** function adds all the functions in the subcategory to the list of functions for the parent category.

If you create a separate category for your new functions, you can use the **generator_add_subcategory** function to add the new category as a subcategory of the relevant Context Sensitive category.

The syntax of **generator_add_subcategory** is as follows:

generator_add_subcategory (*category_name*, *subcategory_name* **);**

➤ *category_name* is the name of an existing category in the Function Generator.

➤ *subcategory_name* is the name of an existing category in the Function Generator.

To add a subcategory to a category:

1. Open the Function Generator. (Choose **Insert** > **Function** > **From Function Generator**, click the **Insert Function from Function Generator** button on the User toolbar, or press the INSERT FUNCTION FROM FUNCTION GENERATOR softkey.)
2. In the **Category** box, click **function table**.
3. In the **Function Name** box, click **generator_add_subcategory**.
4. Click **Args**. The Function Generator expands.
5. In the **Category Name** box, enter the category name as it already appears in the Function Generator.
6. In the **Subcategory Name** box, enter the subcategory name as it already appears in the Function Generator.
7. Click **Paste** to paste the TSL statement into your test script.
8. Click **Close** to close the Function Generator.

A **generator_add_subcategory** statement is inserted into your test script. In the following example, the "my_button" category is defined as a subcategory of the "push_button" category. All "my_button" functions are added to the list of functions defined for the push_button category.

```
generator_add_subcategory ("push_button", "my_button");
```

Note: You must run the test script in order to add a subcategory to a category.

Setting a Default Function for a Category

You set the default function for a category using the **generator_set_default_function** TSL function. This function has the following syntax:

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

- ➤ *category_name* is an existing category.
- ➤ *function_name* is an existing function.

You can set a default function for a standard category or for a user-defined category that you defined using the **generator_add_category** function. If you do not define a default function for a user-defined category, WinRunner uses the first function in the list as the default function.

Note that the **generator_set_default_function** function performs the same operation as the Set As Default button in the Function Generator dialog box. However, a default function set through the Set As Default checkbox remains in effect during the current WinRunner session only. By adding **generator_set_default_function** statements to your startup test, you can set default functions permanently.

To set a default function for a category:

1. Open the Function Generator. (Choose **Insert** > **Function** > **From Function Generator**, click the **Insert Function from Function Generator** button on the User toolbar, or press the INSERT FUNCTION FROM FUNCTION GENERATOR softkey.)
2. In the **Category** box, click **function table**.
3. In the **Function Name** box, click **generator_set_default_function**.
4. Click **Args**. The Function Generator expands.
5. In the **Category Name** box, enter the category name as it already appears in the Function Generator.
6. In the **Default** box, enter the function name as it already appears in the Function Generator.
7. Click **Paste** to paste the TSL statement into your test script.
8. Click **Close** to close the Function Generator.

A **generator_set_default_function** statement is inserted into your test script. In the following example, the default function of the push button category is changed from **button_check_enabled** to the user-defined "check_my_button_label" function.

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

Note: You must run the test script in order to set a default function for a category.

46

Initializing Special Configurations

By creating *startup tests*, you can automatically initialize special testing configurations each time you start WinRunner.

This chapter describes:

- About Initializing Special Configurations
- Creating Startup Tests
- Sample Startup Test

About Initializing Special Configurations

A startup test is a test script that is automatically run each time you start WinRunner. You can create startup tests that load GUI map files and compiled modules, configure recording, and start the application under test.

You designate a test as a startup test by entering its location in the **Startup test** box in the **General** > **Startup** category in the General Options dialog box. For more information on using the General Options dialog box, see Chapter 41, "Setting Global Testing Options."

Creating Startup Tests

You should add the following types of statements to your startup test:

- **load** statements, which load compiled modules containing user-defined functions that you frequently call from your test scripts.
- **GUI_load** statements, which load one or more GUI map files. This ensures that WinRunner recognizes the GUI objects in your application when you run tests.
- statements that configure how WinRunner records GUI objects in your application, such as **set_record_attr** or **set_class_map.**
- an **invoke_application** statement, which starts the application being tested.
- statements that enable WinRunner to generate custom record TSL functions when you perform operations on custom objects, such as **add_cust_record_class**.

By including the above elements in a startup test, WinRunner automatically compiles all designated functions, loads all necessary GUI map files, configures the recording of GUI objects, and loads the application being tested.

Note: You can use the RapidTest Script wizard to create a basic startup test called *myinit* that loads a GUI map file and the application being tested. Note that when you work in the *GUI Map File per Test* mode (described in Chapter 6, "Working in the GUI Map File per Test Mode,") the *myinit* test does not load GUI map files.

Sample Startup Test

The following is an example of the types of statements that might appear in a startup test:

```
# Start the Flight application if it is not already displayed on the screen
if ((rc=win_exists("Flight")) == E_NOT_FOUND)
    invoke_application("w:\\flight_app\\flight.exe", "", "w:\\flight_app",
SW_SHOW);

# Load the compiled module "qa_funcs"
load("qa_funcs", 1, 1);

# Load the GUI map file "flight.gui"
GUI_load ("w:\\qa\\gui\\flight.gui");

# Map the custom “borbtn” class to the standard “push_button” class
set_class_map (“borbtn”, “push_button”);
```

Part VIII

Working with Other Mercury Interactive Products

47

Integrating with QuickTest Professional

You can design tests in QuickTest Professional and then leverage your investments in existing WinRunner script libraries by calling WinRunner tests and functions from your QuickTest test. You can also call QuickTest tests from WinRunner.

This chapter describes calling QuickTest tests from WinRunner. For information on calling WinRunner tests and functions from QuickTest, refer to the *QuickTest Professional User's Guide*.

This chapter describes:

- About Integrating with QuickTest Professional
- Calling QuickTest Tests
- Viewing the Results of a Called QuickTest Test

About Integrating with QuickTest Professional

If you have QuickTest Professional 6.0 or later installed on your computer, you can include calls to QuickTest tests from your WinRunner test. If you have QuickTest Professional 6.5, you can call QuickTest tests and view detailed results of the test call.

You can view the detailed results of the QuickTest test run in the Unified report view of the WinRunner Test Results Window.

When WinRunner runs a called QuickTest test, it automatically loads the QuickTest add-ins required for the test, according to the associated add-ins list specified in the **Properties** tab of the QuickTest Test Settings dialog box.

Note: You cannot call QuickTest tests that use QuickTest's Web Add-in from a WinRunner test if the WebTest Add-in is loaded.

For more information on working with QuickTest Add-ins, refer to the *QuickTest Professional User's Guide*.

When WinRunner is connected to a TestDirector project that contains QuickTest tests, you can call a QuickTest test that is stored in that TestDirector project.

For information on creating QuickTest tests, refer to your QuickTest Professional documentation.

Calling QuickTest Tests

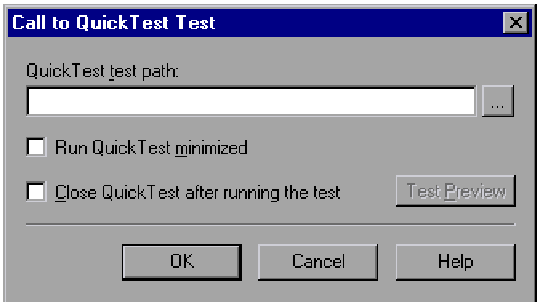
When WinRunner links to QuickTest to run a test, it starts QuickTest, opens the test (in minimized or displayed mode), and runs it. Detailed information about the results of the QuickTest test run are displayed in the Unified report view of the WinRunner Test Results window.

You can insert a call to a QuickTest test using the Call to QuickTest Test dialog box or by manually entering a **call_ex** statement.

Note: You cannot call a QuickTest test that includes calls to WinRunner tests.

To insert a call to a QuickTest test using the Call to QuickTest Test dialog box:

1 Choose **Insert** > **Call to QuickTest Test**. The Call to QuickTest Test dialog box opens.



2 In the **QuickTest test path** box, enter the path of the QuickTest test or browse to it.

If you are connected to TestDirector when you click the browse button, the Open from TestDirector project dialog box opens so that you can select the test from the TestDirector project. For more information on this dialog box, see Chapter 48, “Managing the Testing Process.”

3 Select **Run QuickTest minimized** if you do not want to view the QuickTest window while the test runs. (This option is supported only for QuickTest 6.5 or later.)

4 Select **Close QuickTest after running the test** if you want the QuickTest application to close when the step calling the QuickTest test is complete.

5 You can click **Test Preview** to view an outline of all actions in the QuickTest test or to view the complete Expert View script of any selected action.

If you are connected to TestDirector, a message box may open informing you that downloading the QuickTest test from TestDirector may take some time. Click **OK** to download the QuickTest test. You can also clear the **Show this message next time** check box if you do not want to see this message again.

The Test Preview dialog box opens.

6 Preview the QuickTest test details and click **OK** in the Test Preview dialog box to close it.

7 Click **OK** to close the dialog box. A **call_ex** statement similar to the following is inserted in your test:

```
call_ex("F:\\Merc_Progs\\QTP\\Tests\\web\\short_flight",1,1);
```

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

call_ex (*QT_test_path* [**,** *run_minimized***,** *close_QT*] **);**

Note: The **call_ex** statement provided with WinRunner 7.5 returned different values than the 7.6 version of this function. If you have tests that were created in WinRunner 7.5 and use the return value of this function, you may need to modify your test to reflect the new return values. For more information on these methods, refer to the *WinRunner TSL Reference*.

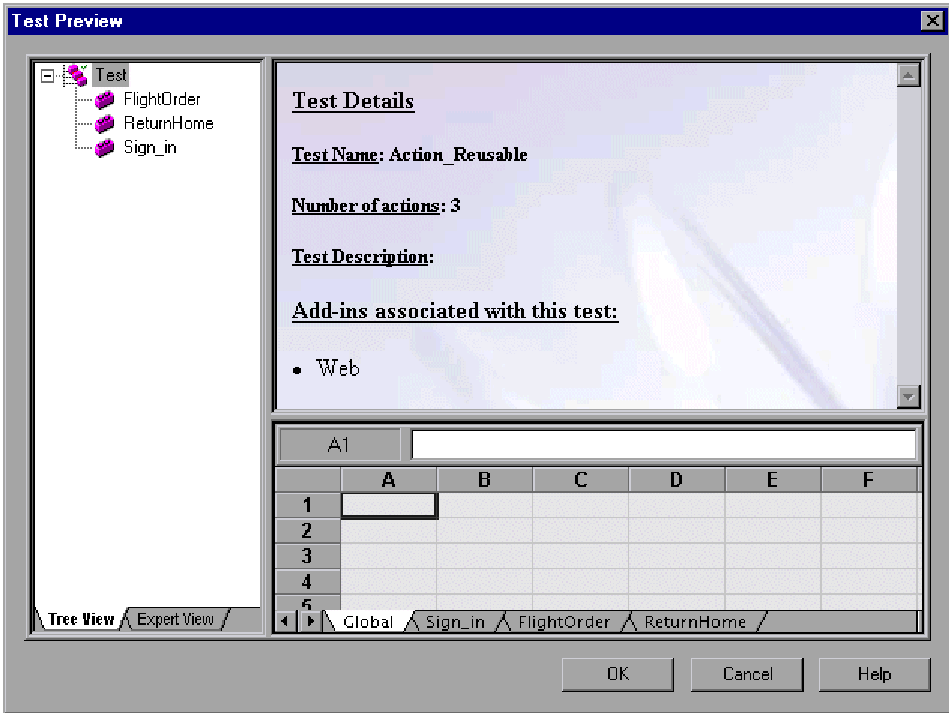
For additional information on the **call_ex** function and an example of usage, refer to the *WinRunner TSL Reference*.

Previewing a QuickTest Test

Before you insert a call_ex statement in your test using the Call to QuickTest Test dialog box, you can preview a selected QuickTest test.

To do so, open the Call to QuickTest Test dialog box and select a test as described in "Calling QuickTest Tests" on page 906. Then click the **Test Preview** button.

The **Test Preview** window opens.



Test Details

Test Name: Action_Reusable

Number of actions: 3

Test Description:

Add-ins associated with this test:

- Web

The Test Preview window enables you to preview the QuickTest test before you insert a call to it into your WinRunner test. You cannot edit the QuickTest from this window.

The Test Preview window is divided into 3 panes:

- **Test pane**—Displays a preview of the test in Tree View and Expert View formats. The Tree View pane displays an outline of the actions contained in the test. The Expert View pane displays the complete script (VBScript) of the selected action.
- **Details pane**—When the Tree View tab is selected, the Details pane displays summary information about the test or selected action. When the Expert View tab is selected, the Details pane displays the captured Active Screen for the selected line in the test.

- **Data Table pane**—Displays the test's design-time data table, containing the data used for the test's data table parameters. You can view the Global data sheet or any action sheet.

Viewing the Results of a Called QuickTest Test

You can view the results of any WinRunner test run in the WinRunner report view or the unified report view. However, to view detailed information about a called QuickTest 6.5 test, you must ensure that WinRunner is set to generate unified report information before you run your test, and that it is set to display the unified report when you view your test results.

To instruct WinRunner to create unified report information and display the unified report:

1. Before running your test (or before displaying the test results), choose **Tools** > **General Options**. The General Options dialog box opens.
2. Click the **Run** category.
3. To ensure that unified report information is created before a test run, select **Unified report view** or select **WinRunner report view** and the **Generate unified report information** option.

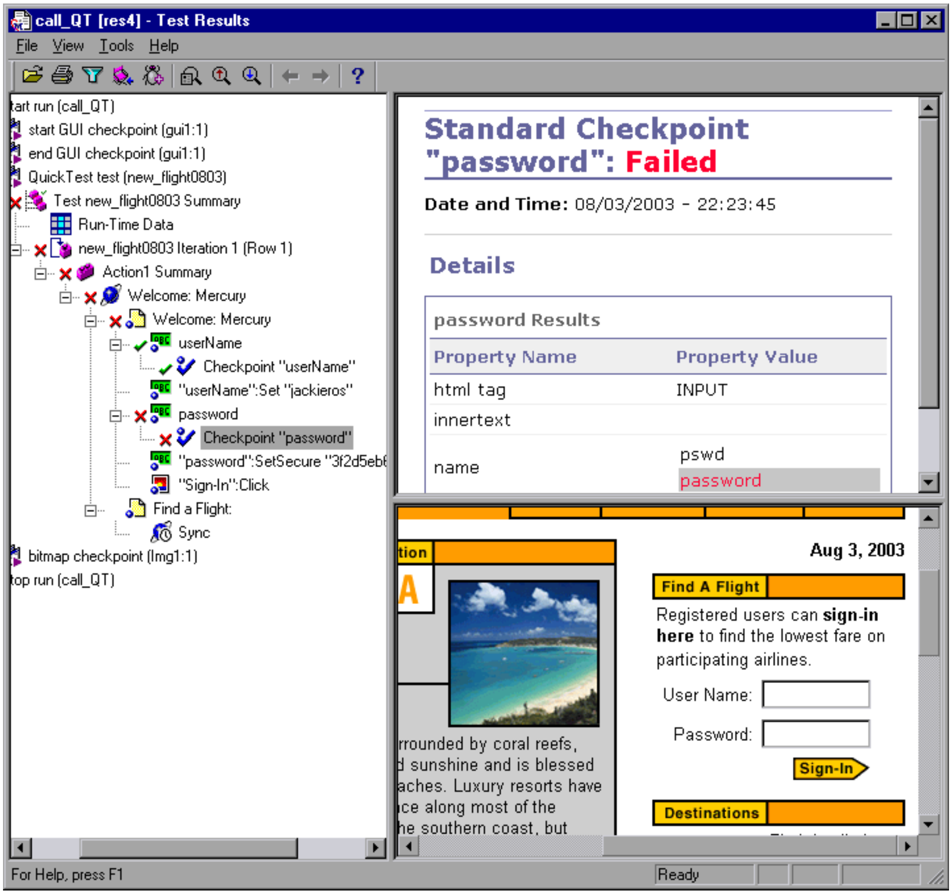
   To display the unified report information, select **Unified report view** before opening the Test Results window.

   For more information, see Chapter 34, "Analyzing Test Results."

Analyzing the Results of a Called QuickTest Test

The unified report view of the WinRunner Test Results window includes a node for each event in your WinRunner test, plus a node for each step of the called QuickTest test.

When you select a node corresponding to a QuickTest step, the right pane displays details of the step and may contain a screen capture of the application at the time the step was performed.



tart run (call_QT)
start GUI checkpoint (gui1:1)
end GUI checkpoint (gui1:1)
QuickTest test (new_flight0803)
Test new_flight0803 Summary
Run-Time Data
new_flight0803 Iteration 1 (Row 1)
Action1 Summary
Welcome: Mercury
Welcome: Mercury
userName
Checkpoint "userName"
"userName":Set "jackieros"
password
Checkpoint "password"
"password":SetSecure "3f2d5ebf
"Sign-In":Click
Find a Flight:
Sync
bitmap checkpoint (Img1:1)
top run (call_QT)

Standard Checkpoint "password": Failed

Date and Time: 08/03/2003 - 22:23:45

Details

password Results

Property Name	Property Value
html tag	INPUT
innertext	
name	pswd password

tion

A

rrounded by coral reefs,
d sunshine and is blessed
aches. Luxury resorts have
ce along most of the
he southern coast, but

Aug 3, 2003

Find A Flight

Registered users can **sign-in here** to find the lowest fare on participating airlines.

User Name:

Password:

Sign-In

Destinations

Note: You can view the results of the called QuickTest test only in the WinRunner unified report view from the results folder of the WinRunner test. The results of the QuickTest test are not saved under the called QuickTest test folder.

When analyzing the results of a WinRunner test containing a call to a QuickTest test, you may want to view the following:

- Select the **start run** node to view summary results of the WinRunner test. This summary indicates the status of the entire test run, but includes summary checkpoint information only for the WinRunner steps in your test.
- Select a **WinRunner** node to view the results of WinRunner events, just as you would with any WinRunner test.
- Select the QuickTest **Test** node to view summary results of the called QuickTest test. This summary includes the status of the QuickTest test run, and statistical information about the checkpoints contained in the QuickTest test.
- Select the QuickTest **Run-Time Data** node to view the resulting Data Table of the QuickTest test, including data used in Data Table parameters and data stored in the table during the test run by output values in the test.
- Select an **iteration** node to view summary information for a test iteration.
- Select an **action** node to view summary information for an action.
- Select a QuickTest step node to view detailed information about the results of the selected step. If a screen was captured for the selected step, the captured screen is displayed in the bottom right pane of the Test Results window.

  By default, QuickTest only captures screens for failed steps. You can change the **Save step screen capture to test results** option in the **Run** tab of the QuickTest Options dialog box.

  For more information on the data provided for various QuickTest test steps, refer to the *QuickTest Professional User's Guide*.

  For more information on analyzing WinRunner Test Results, see "Analyzing Test Results" on page 643.

48

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

TestDirector is a powerful test management tool that helps you systematically control the testing process. It helps you create a framework and foundation for your testing workflow.

TestDirector helps you maintain a project of tests that cover all aspects of your application's functionality. Every test in your project is designed to fulfill a specified testing requirement of your application. To meet the goals of a project, you organize the tests in your project into unique groups. TestDirector provides an intuitive and efficient method for scheduling and running tests, collecting test results, and analyzing the results.

It also features a system for tracking defects, enabling you to monitor defects closely from initial detection until resolution.

WinRunner works with TestDirector 7.x and 8.0.

TestDirector versions 7.5 and later provide version control support, which enables you to update and revise your automated test scripts while maintaining old versions of each test. This helps you keep track of the changes made to each test script, see what was modified from one version of a script to another, or return to a previous version of the test script. For more information on version control support, see "Managing Test Versions in WinRunner" on page 924.

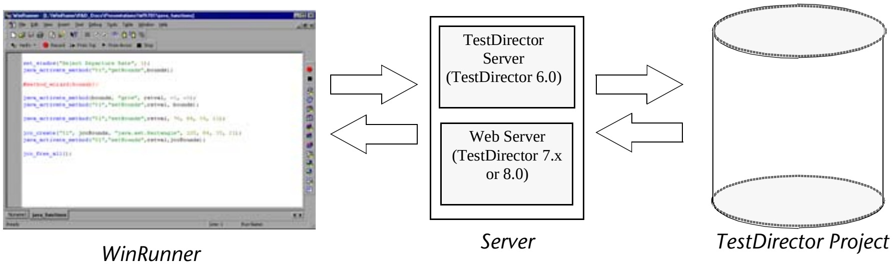
Note: This chapter describes the integration of WinRunner with TestDirector 7.x and 8.0. For more information on working with TestDirector, refer to the *TestDirector User's Guide*.

Integrating the Testing Process

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 you run your test, you can view and analyze the results in TestDirector.

TestDirector stores test and defect information in a project. You can create TestDirector projects in Microsoft Access, Oracle, Sybase, or Microsoft SQL. These projects store information related to the current testing project, such as tests, test run results, and reported defects.

In order for WinRunner to access the project, you must connect it to the Web server where TestDirector is installed.


TestDirector Server (TestDirector 6.0)
Web Server (TestDirector 7.x or 8.0)
WinRunner
Server
TestDirector Project

When WinRunner is connected to TestDirector, you can save a test by associating it with the Test Plan Manager. You can schedule to run a test on local or remote hosts. Test run results are sent directly to your TestDirector project.

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 **Run** category of the General Options dialog box (**Tools** > **General Options**). For more information on setting this option, see Chapter 41, “Setting Global Testing Options.”

Accessing WinRunner Tests from TestDirector 7.x and 8.0

When TestDirector accesses a WinRunner test, the test is downloaded from a project database to a local temporary directory, which becomes your current working directory. If the test calls another file (for example, a module or a test), and the full pathname of the called file is not specified, the current working directory becomes the relative path of the referenced file. Therefore, WinRunner cannot open the called test.

For example, suppose a test calls the flt_lib file:

```
static lib_path = getvar("testname") & "\\..\\flt_lib";
reload(lib_path);
```

WinRunner looks for the called test in the relative path. To enable WinRunner to find the correct pathname, you can:

- change the pathname of the WinRunner called file, or
- set direct file access for all WinRunner tests (LAN only)

Changing the Pathname of Files

To enable WinRunner to access a called file from a test, save the file in your TestDirector project and then change the pathname in your WinRunner test script.

For example, suppose you save the flt_lib file in your TestDirector project under subject\\module. TestDirector now calls the file using the following statement:

```
static lib_path = "[TD]\\Subject\\module\\flt_lib";
```

For more information on saving tests to a TestDirector project, see “Saving Tests to a Project” on page 921.

Accessing WinRunner Tests Directly (LAN only)

If you are working in a local area network (LAN) environment, you can set your machine so that it provides direct file access to all WinRunner tests, regardless of their directory path. This enables you to run WinRunner tests from TestDirector without changing the directory path of other called tests.

To set the direct file access option:

1. On the machine where WinRunner is installed, click **Run** on the **Start** menu. The Run dialog box opens.
2. Type regedit and click **OK**. The Registry Editor opens.
3. Locate the following folder:

   My Computer > **HKEY_LOCAL_MACHINE** > **Software** > **Mercury Interactive** > **TestDirector** > **Testing Tools** > **WinRunner**.
4. In the **WinRunner** folder, double-click **DirectFileAccess**. Change the value in the Value Data box to "Y".

Tip: After setting the direct access option, your Web access performance will improve while accessing WinRunner tests from TestDirector.

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.

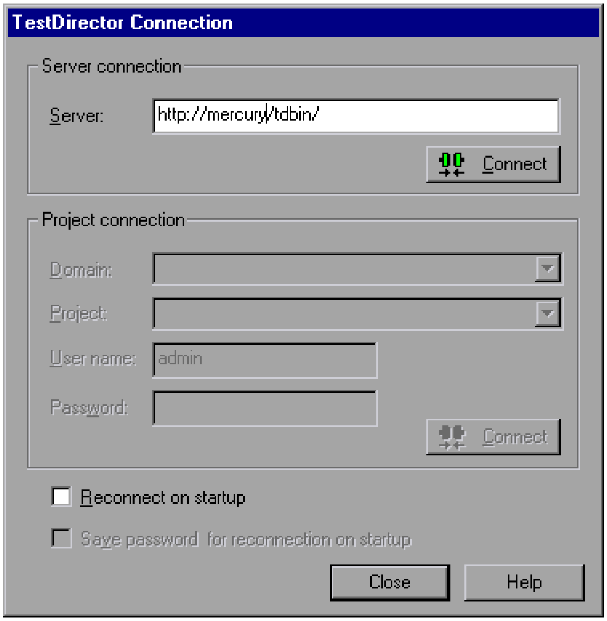
The connection process has two stages. First, you connect WinRunner to the TestDirector server. This server handles the connections between WinRunner and the TestDirector project. Next, you choose the project you want WinRunner to access. The project stores tests and test run information for the application you are testing. Note that TestDirector projects are password protected, so you must provide a user name and a password.

Connecting WinRunner to TestDirector

You must connect WinRunner to the server before you connect WinRunner to a project. For more information, see "Integrating the Testing Process" on page 915.

To connect WinRunner to TestDirector:

1 Choose **Tools** > **TestDirector Connection**. The TestDirector Connection dialog box opens.



TestDirector Connection

Server connection

Server: http://mercury/tdbin/

Connect

Project connection

Domain:

Project:

User name: admin

Password:

Connect

☐ Reconnect on startup

☐ Save password for reconnection on startup

2 In the **Server** box, type the URL of the Web server where TestDirector is installed.

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 If you are connecting to a project in TestDirector 7.5 or later, enter or select the domain that contains the TestDirector project, in the **Domain** box.

5 Enter the TestDirector project name or select a project from the **Project** list.

6 In the **User name** box, type a user name.

7 In the **Password** box, type a password.

8 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.

Note: If **Reconnect on startup** is selected, but you want to open WinRunner without connecting to TestDirector, you can use the *-dont_connect* command line option as described in Chapter 30, "Running Tests from the Command Line."

---

9 Click **Close** to close the TestDirector Connection 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. For more information on these options, see "Command Line Options for Working with TestDirector," on page 937. For more information on using command line options, see Chapter 36, "Running Tests from the Command Line."

Disconnecting from a TestDirector Project

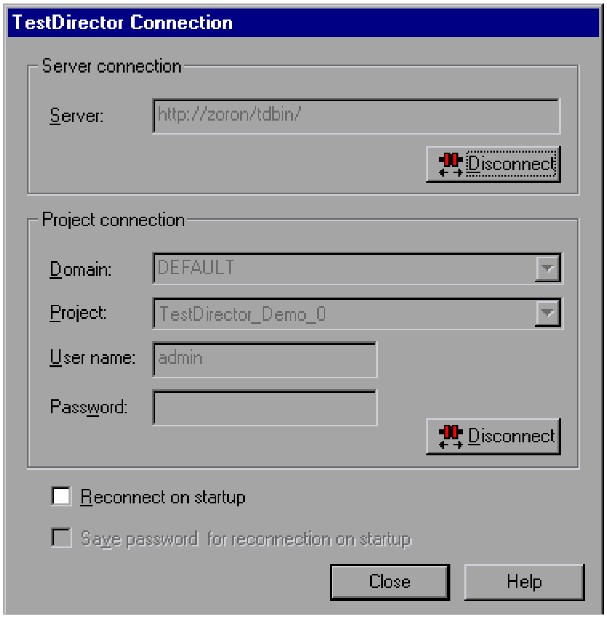
You can disconnect from a TestDirector project or server. Note that if you disconnect WinRunner from a server without first disconnecting from a project, WinRunner's connection to that database is automatically disconnected.

Note: When disconnecting from TestDirector, if a test is opened from TestDirector, then WinRunner closes it.

To disconnect WinRunner from a project:

1 Choose **Tools** > **TestDirector Connection**.

The TestDirector Connection dialog box opens.



TestDirector Connection

Server connection

Server: http://zoron/tdbin/

Project connection

Domain: DEFAULT

Project: TestDirector_Demo_0

User name: admin

Password:

□ Reconnect on startup

□ Save password for reconnection on startup

2 In the **Project connection** section, click **Disconnect** to disconnect WinRunner from the selected project. If you want to open a different project while using the same server, select the project as described in step 5 on page 918.

3 To disconnect WinRunner from the TestDirector server, click **Disconnect** in the **Server connection** section.

4 Click **Close** to close the TestDirector Connection dialog box.

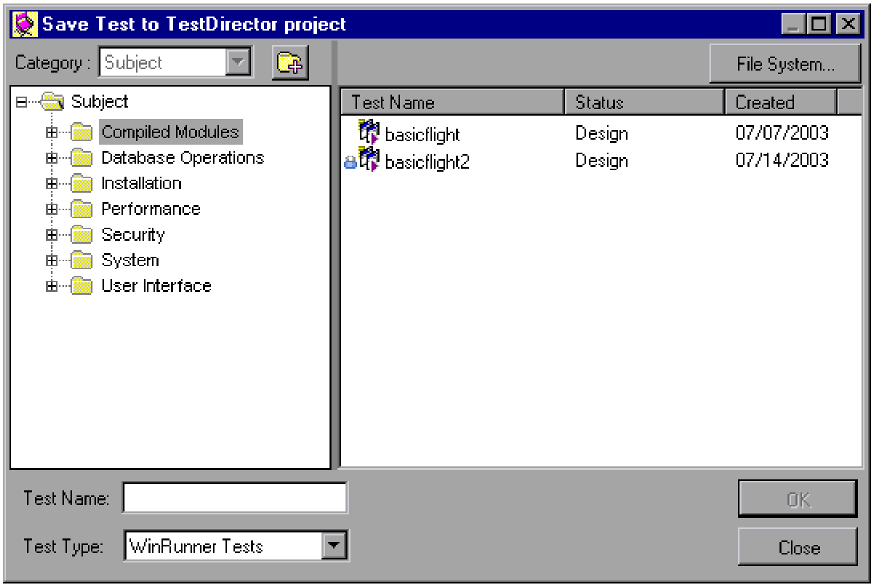
Saving Tests to a Project

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.



Category : Subject

Test Name	Status	Created
basicflight	Design	07/07/2003
basicflight2	Design	07/14/2003

Test Name:

Test Type: WinRunner Tests

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** box, enter a name for the test. Use a descriptive name that will help you easily identify the test.
4. Click **OK** to save the test and close the dialog box.

Note: To save a batch test, choose **WinRunner Batch Tests** in the **Test Type** box.

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

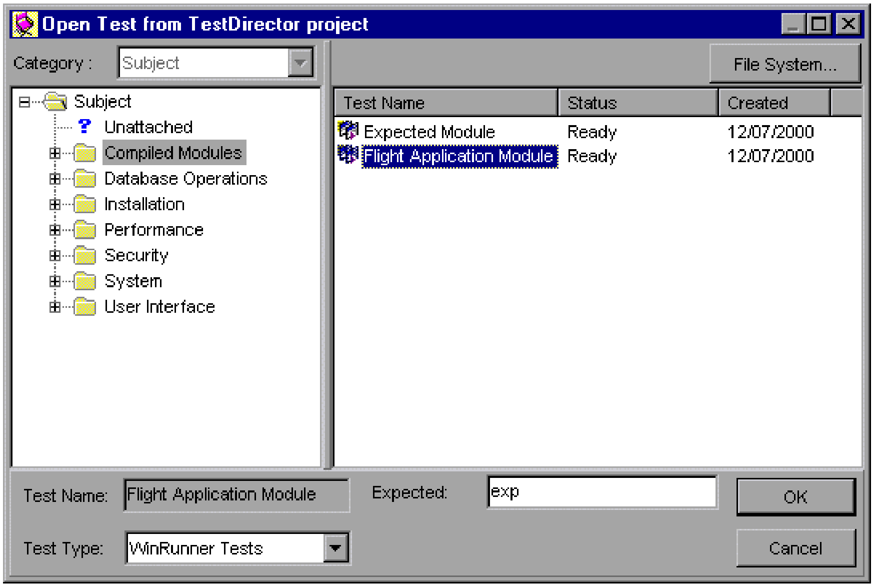
Opening Tests in a Project

If WinRunner is connected to a TestDirector project, you can open automated tests that are a part of your project. You locate tests according to their position in the test plan tree, rather than by their actual location in the file system.

To open a test saved to a 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.



Category : Subject

Test Name	Status	Created
Expected Module	Ready	12/07/2000
Flight Application Module	Ready	12/07/2000

Test Name: Flight Application Module

Expected: exp

Test Type: WinRunner Tests

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

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 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 35, "Running Batch Tests."

Managing Test Versions in WinRunner

When WinRunner is connected to a TestDirector (versions 7.5 or later only) 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 about the TestDirector version control add-ins, refer to your *TestDirector Installation Guide*.

You manage test versions by checking tests in and out of the version control database.

Adding Tests to the Version Control Database

When you add a test to the version control database for the first time, it becomes the *Working Test* and is also assigned a permanent version number.

The working test is the test that is located in the test repository and is used by TestDirector for all test runs.

Note: Usually the latest version is the working test, but any version can be designated as the working test in TestDirector. For more information, refer to your TestDirector documentation.

To add a new test to the version control database:

1 Choose **File** > **Check In.**

Note: The Check In and Check Out options in the File menu are visible only when you are connected to a TestDirector project database with version control support, and you have a test open. The Check In option is enabled only if the active script has been saved to the project database.

2 Click **OK** to confirm adding the test to the version control database.

3 Click **OK** to reopen the checked-in test. The test will close and then reopen as a read-only file.

If you have made unsaved changes in the active test, you will be prompted to save the test.

You can review the checked-in test. You can also run the test and view the results. While the test is checked in and is in read-only format, however, you cannot make any changes to the script.

If you attempt to make changes, a WinRunner message reminds you that the script has not been checked out and that you cannot change it.

Checking Tests Out of the Version Control Database

When you open a test that is currently checked in to the version control database, you cannot make any modifications to the script. If you wish to make modifications to this script, you must check out the script.

When you check out a test, TestDirector 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.

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, TestDirector 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 TestDirector to change the checked in/checked out status of the test. If you make changes to the test's status using TestDirector while the test is open in WinRunner, WinRunner will not reflect those changes. For more information, refer to your TestDirector documentation.

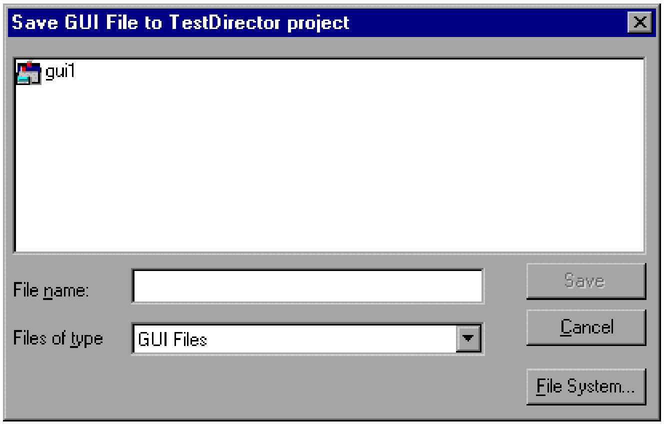
Saving GUI Map Files to a Project

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.

Note: When you choose to save a GUI map file to a TestDirector project, it is uploaded to the project immediately.

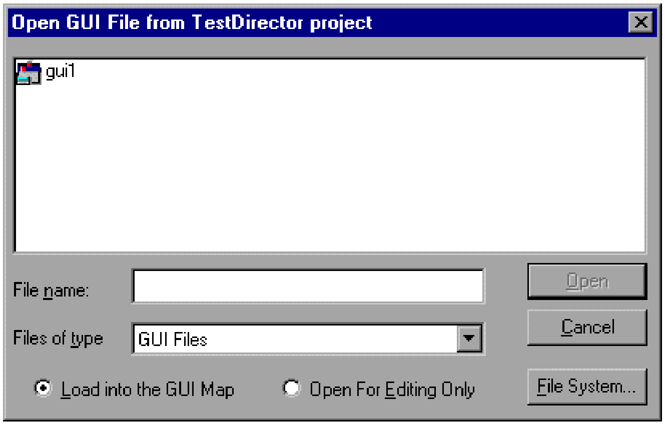
Opening GUI Map Files in a Project

When WinRunner is connected to a TestDirector project, you can use the GUI Map Editor to open a GUI map file saved to a TestDirector project.

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

File name:

Files of type: GUI Files

● Load into the GUI Map ○ Open For Editing Only

Note that the Open GUI File from TestDirector project dialog box opens only when WinRunner is connected to a TestDirector project.

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 7, “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

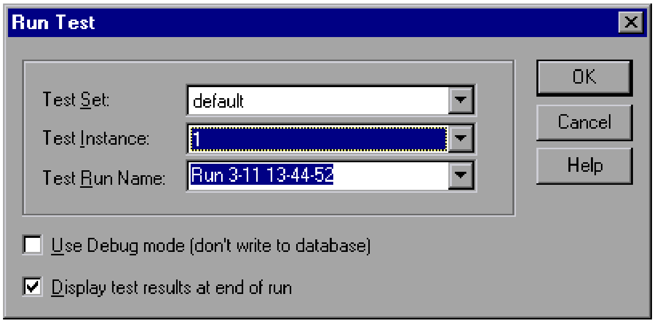
A test set is a group of tests selected to achieve specific testing goals. For example, you can create a test set that tests the user interface of the application or the application's performance under stress. You define test sets when working in TestDirector's test run mode.

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 from the **Test** menu.

The Run Test dialog box opens.


Run Test

Test Set: default

Test Instance: 1

Test Run Name: Run 3-11 13-44-52

□ 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 If you are working with TestDirector 7.5 or later, and your test set contains more than one instance of the test, select the **Test Instance**. If you are working with an earlier version of TestDirector, the Test Instance is always 1.

4 In the **Test Run Name** box, select a name for this test run, or enter a new name.

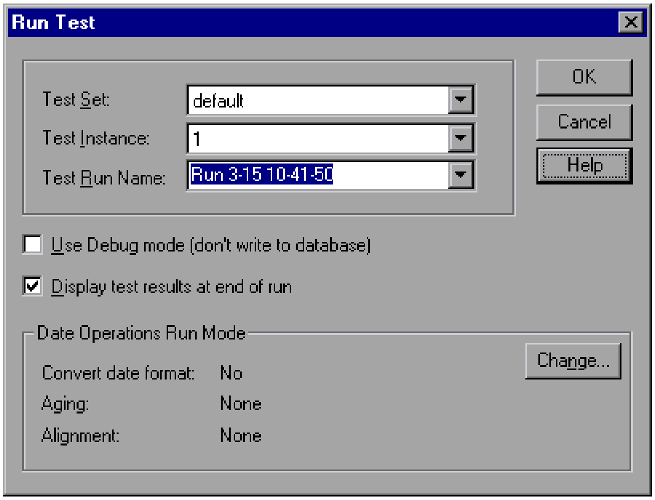
To run tests in Debug mode, select the **Use Debug mode** check box. If this option is selected, the results of this test run are not written to the TestDirector project.

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.

5 Click **OK** to save the parameters and to run the test.

Running Date Operations Tests in a Test Set

If the **Enable date operations** option is selected (**Tools** > **General Options** > **General** category), you can also view and modify the Date Operations Run Mode settings from the Run Test dialog box.



Run Test

Test Set: default

Test Instance: 1

Test Run Name: Run 3-15 10-41-50

□ Use Debug mode (don't write to database)

☑ Display test results at end of run

Date Operations Run Mode

Convert date format: No

Aging: None

Alignment: None

For more information on running tests to check date operations, see “Running a Test to Check Date Operations,” on page 633.

Running Tests on Remote Hosts

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 44, "Setting Testing Options from a Test Script."

To enable TestDirector on a remote machine to run WinRunner tests:

1. Choose **Tools** > **General Options** to open the General Options dialog box.
2. Click the **Run** category.
3. Select the **Allow 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 41, "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.

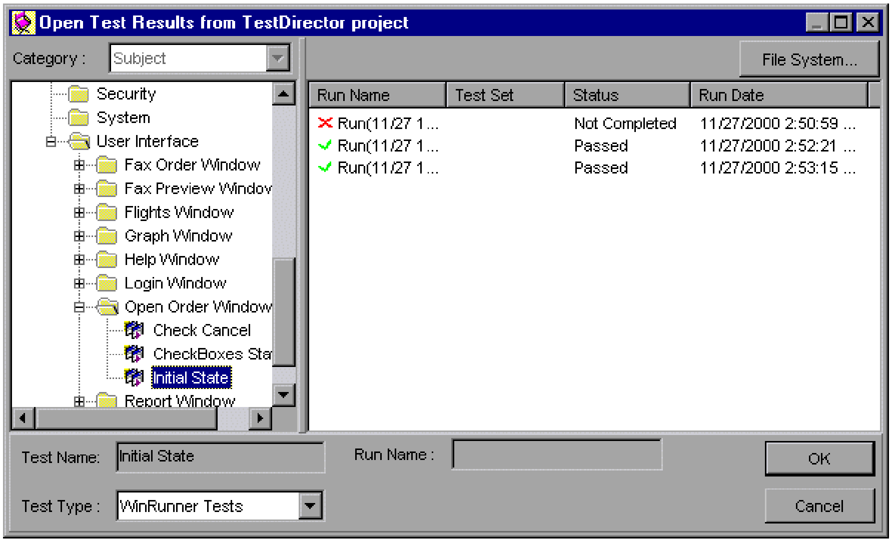
To view test results from a TestDirector project:

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

Run Name	Test Set	Status	Run Date
✗ Run(11/27 1...		Not Completed	11/27/2000 2:50:59 ...
✓ Run(11/27 1...		Passed	11/27/2000 2:52:21 ...
✓ Run(11/27 1...		Passed	11/27/2000 2:53:15 ...

Test Name: Initial State

Run Name :

Test Type : WinRunner Tests

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.

If the test results indicate defects in your application, you can report the defects to your TestDirector defect database directly from the Test Results window. For more information, see "Reporting Defects Detected During a Test Run" on page 687.

For information about the options in the Test Results window, see Chapter 34, "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_add_defect	Adds a new defect to the TestDirector defect database for the project to which WinRunner is connected.
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.
tddb_load_attachment	Downloads a file attachment of a test to the local cache and returns its location.

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

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

Call Statements and Compiled Module Functions

When WinRunner is connected to 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 **Folders** category of the General Options dialog box or by using a **setvar** statement in your test script, you can call the test from your test script with the following statement:

```
call "My_test" ();
```

Note that the [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 Reference*.

GUI Map Editor Functions

When WinRunner is connected to TestDirector, you can specify the names of GUI map files saved in a TestDirector project 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 Reference*.

Specifying Search Paths for Tests Called from TestDirector

You can configure WinRunner to use search paths based on the path in a TestDirector project.

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 41, "Setting Global Testing Options." For information on how to specify the search path by using a **setvar** statement, see Chapter 44, "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 command line options described below to set parameters for working with TestDirector. For additional information on using command line options, see Chapter 36, "Running Tests from the Command Line."

-dont_connect

If the **Reconnect on startup** check box is selected in the TestDirector Connection dialog box, this command line enables you to open WinRunner without connecting to TestDirector.

-td_connection {on | off}

Activates WinRunner's connection to TestDirector when set to **on**.

(Default = **off**)

(Formerly **-test_director.**)

Note: If you select the "Reconnect on startup" option in the Connection to Test Director dialog box, setting **-td_connection** to off will not prevent the connection to TestDirector. To prevent the connection to TestDirector in this situation, use the **-dont_connect** command. For more information, see "-dont_connect," on page 937.

-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 44, "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 44, "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 917.

-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 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 917.

-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 44, "Setting Testing Options from a Test Script."

Note that you can specify the name of the TestDirector server to which WinRunner connects 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 917.

-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 44, "Setting Testing Options from a Test Script."

Note that you can specify the user name when you connect to TestDirector 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 917.

For more information on using command line options, see Chapter 36, "Running Tests from the Command Line."

49

Testing Systems Under Load

Today's applications are run by multiple users over complex architectures. With LoadRunner, TestSuite's load testing tool, you can test the performance and reliability of an entire system.

This chapter describes:

- About Testing Systems Under Load
- Emulating Multiple Users
- Virtual User (Vuser) Technology
- Developing and Running Scenarios
- Creating GUI Vuser Scripts
- Measuring Server Performance
- Synchronizing Virtual User Transactions
- Creating a Rendezvous Point
- A Sample Vuser Script

About Testing Systems Under Load

Software testing is no longer confined to testing applications that run on a single, standalone PC. Applications are run in network environments where multiple client PCs or UNIX workstations interact with a central server. Web-based applications are also common.

Modern architectures are complex. While they provide an unprecedented degree of power and flexibility, these systems are difficult to test. LoadRunner emulates load and then accurately measures and analyzes performance and functionality. This chapter provides an overview of how to use WinRunner together with LoadRunner to test your system. For detailed information about how to load test an application, refer to your LoadRunner documentation.

Emulating Multiple Users

With LoadRunner, you emulate the interaction of multiple users by creating *scenarios*. A scenario defines the events that occur during each load testing session, such as the number of users, the actions they perform, and the machines they use. For more information about scenarios, refer to the *LoadRunner Controller User's Guide*.

In the scenario, LoadRunner replaces the human user with a *virtual user or Vuser.* A Vuser emulates the actions of a human user working with your application. A scenario can contain tens, hundreds, or thousands of Vusers.

Virtual User (Vuser) Technology

LoadRunner provides a variety of Vuser technologies that enable you to generate load when using different types of system architectures. Each Vuser technology is suited to a particular architecture, and results in a specific type of Vuser. For example, you use GUI Vusers to operate graphical user interface applications in environments such as Microsoft Windows; Web Vusers to emulate users operating Web browsers; RTE Vusers to operate terminal emulators; Database Vusers to emulate database clients communicating with a database application server.

The various Vuser technologies can be used alone or together, to create effective load testing scenarios.

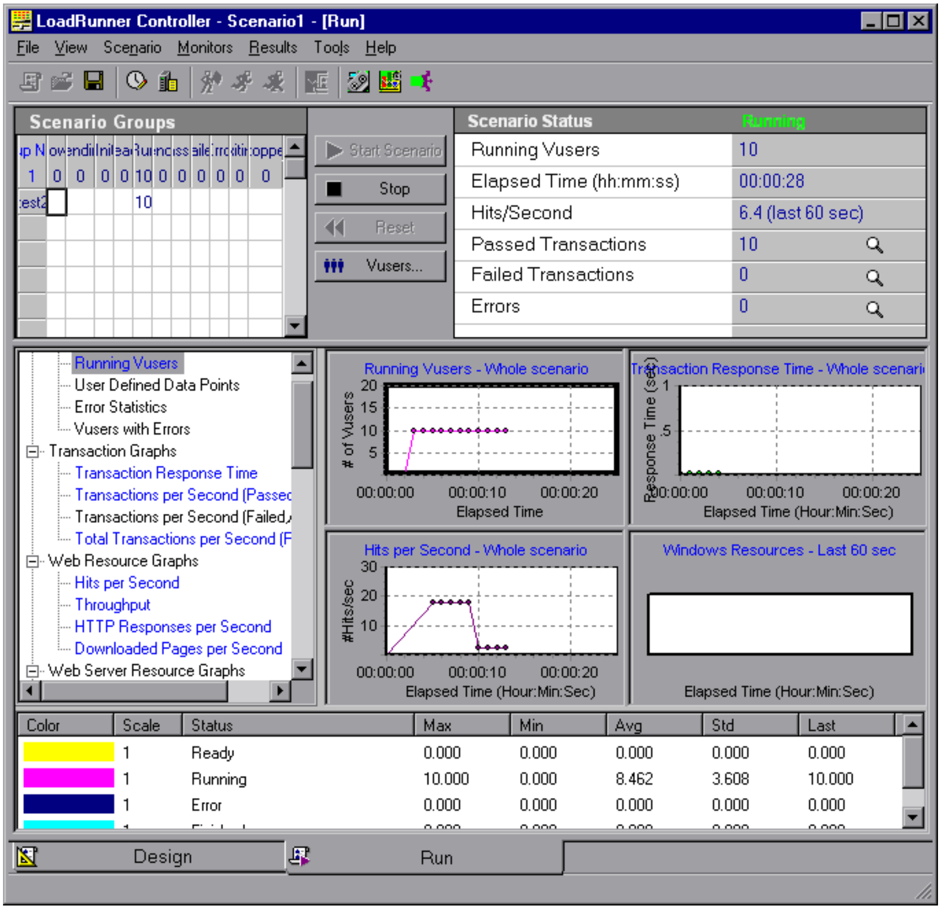
GUI Vusers

GUI Vusers operate graphical user interface applications in environments such as Microsoft Windows. Each GUI Vuser emulates a real user submitting input to and receiving output from a client application.

A GUI Vuser consists of a copy of WinRunner and a client application. The client application can be any application used to access the server, such as a database client. WinRunner replaces the human user and operates the client application. Each GUI Vuser executes a Vuser script. This is a WinRunner test that describes the actions that the Vuser will perform during the scenario. It includes statements that measure and record the performance of the server. For more information, refer to the LoadRunner *Creating Vuser Scripts* guide.

Developing and Running Scenarios

You use the LoadRunner Controller to develop and run scenarios. The Controller is an application that runs on any network PC.


LoadRunner Controller - Scenario1 - [Run]
File View Scenario Monitors Results Tools Help
Scenario Groups
Start Scenario
Stop
Reset
Vusers...
Running Vusers
User Defined Data Points
Error Statistics
Vusers with Errors
Transaction Graphs
Transaction Response Time
Transactions per Second (Passed
Transactions per Second (Failed,
Total Transactions per Second (F
Web Resource Graphs
Hits per Second
Throughput
HTTP Responses per Second
Downloaded Pages per Second
Web Server Resource Graphs
Running Vusers - Whole scenario
# of Vusers
20
15
10
5
00:00:00
00:00:10
00:00:20
Elapsed Time
Transaction Response Time - Whole scenari
Response Time (sec)
1
.5
00:00:00
00:00:10
00:00:20
Elapsed Time (Hour:Min:Sec)
Hits per Second - Whole scenario
#Hits/sec
30
20
10
00:00:00
00:00:10
00:00:20
Elapsed Time (Hour:Min:Sec)
Windows Resources - Last 60 sec
Elapsed Time (Hour:Min:Sec)
Design
Run

Scenario Status	Running
Running Vusers	10
Elapsed Time (hh:mm:ss)	00:00:28
Hits/Second	6.4 (last 60 sec)
Passed Transactions	10
Failed Transactions	0
Errors	0

Color	Scale	Status	Max	Min	Avg	Std	Last
🟨	1	Ready	0.000	0.000	0.000	0.000	0.000
🟪	1	Running	10.000	0.000	8.462	3.608	10.000
🟦	1	Error	0.000	0.000	0.000	0.000	0.000

The following procedure outlines how to use the LoadRunner Controller to create, run, and analyze a scenario. For more information, refer to the *LoadRunner Controller User's Guide*.

1 **Invoke the Controller.**

2 **Create the scenario.**

A scenario describes the events that occur during each load testing session, such as the participating Vusers, the scripts they run, and the machines the Vusers use to run the scripts (load generating machines).

3 **Run the scenario.**

When you run the scenario, LoadRunner distributes the Vusers to their designated load generating machines. When the load generating machines are ready, they begin executing the scripts. During the scenario run, LoadRunner measures and records server performance data, and provides online network and server monitoring.

4 **Analyze server performance.**

After the scenario 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.

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 22, "Synchronizing the Test Run."

Creating a Rendezvous Point

During the scenario run, you instruct multiple Vusers to perform tasks simultaneously by creating a rendezvous point. This ensures that:

- intense user load is emulated
- transactions are measured under the load of multiple Vusers

A rendezvous point is a meeting place for Vusers. To designate the meeting place, you insert rendezvous statements into your Vuser scripts. When the rendezvous statement is interpreted, the Vuser is held by the Controller until all the members of the rendezvous arrive. When all the Vusers have arrived (or a time limit is reached), they are released together and perform the next task in their Vuser scripts.

The first stage of creating a rendezvous point is to declare its name at the start of the Vuser script. When you assign the Vuser script to a Vuser, LoadRunner scans the script for rendezvous declaration statements. If the script contains a rendezvous declaration, LoadRunner reads the rendezvous name and creates a rendezvous. If you create another Vuser that runs the same script, the Controller will add the Vuser to the rendezvous.

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

declare_rendezvous ("*rendezvous_name*" **);**

where *rendezvous_name* is the name of the rendezvous. The *rendezvous_name* must be a string constant, not a variable or an expression. This string can contain up to 128 characters. No spaces are permitted.

Next, you indicate the point in the Vuser script where the rendezvous will occur by inserting a **rendezvous** statement. This tells LoadRunner to hold the Vuser at the rendezvous until all the other Vusers arrive. The function has the following syntax:

rendezvous ("*rendezvous_name*" **)**;

The *rendezvous_name* is the name of the rendezvous.

A Sample Vuser Script

In the following sample Vuser script, the “Ready” transaction measures how long it takes for the server to respond to a request from a user. The user enters the request and then clicks OK. The user knows that the request has been processed when the word “Ready” appears in the client application’s Status text box.

In the first part of the Vuser script, the **declare_transaction** and **declare_rendezvous** functions declare the names of the transaction and rendezvous points in the Vuser script. In this script, the transaction “Ready” and the rendezvous “wait” are declared. The declaration statements enable the LoadRunner Controller to display transaction and rendezvous information.

```
# Declare the transaction name
declare_transaction ("Ready");
```

```
# Define the rendezvous name
declare_rendezvous ("wait");
```

Next, a **rendezvous** statement ensures that all Vusers click OK at the same time, in order to create heavy load on the server.

```
# Define rendezvous points
rendezvous ("wait");
```

In the following section, a **start_transaction** statement is inserted just before the Vuser clicks OK. This instructs LoadRunner to start recording the “Ready” transaction. The “Ready” transaction measures the time it takes for the server to process the request sent by the Vuser.

```
# Deposit transaction
start_transaction ( "Ready" );
button_press ( "OK" );
```

Before LoadRunner can measure the transaction time, it must wait for a cue that the server has finished processing the request. A human user knows that the request has been processed when the "Ready" message appears under Status; in the Vuser script, an **obj_wait_info** statement waits for the message. Setting the timeout to thirty seconds ensures that the Vuser waits up to thirty seconds for the message to appear before continuing test execution.

```
# Wait for the message to appear
rc = obj_wait_info("Status","value","Ready.",30);
```

The final section of the test measures the duration of the transaction. An if statement is defined to process the results of the **obj_wait_info** statement. If the message appears in the field within the timeout, the first **end_transaction** statement records the duration of the transaction and that it passed. If the timeout expires before the message appears, the transaction fails.

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

Index

MERCURY
INTERACTIVE

Mercury Interactive Corporation
1325 Borregas Avenue
Sunnyvale, CA 94089 USA

Main Telephone: (408) 822-5200
Sales & Information: (800) TEST-911, (866) TOPAZ-4U
Customer Support: (877) TEST-HLP
Fax: (408) 822-5300

Home Page: www.mercuryinteractive.com
Customer Support: support.mercuryinteractive.com