



Basic Features User's Guide

MERCURY™

Mercury WinRunner

Basic Features User's Guide

Version 8.2

MERCURY™

Mercury WinRunner Basic Features User's Guide, Version 8.2

This manual, and the accompanying software and other documentation, is protected by U.S. and international copyright laws, and may be used only in accordance with the accompanying license agreement. Features of the software, and of other products and services of Mercury Interactive Corporation, may be covered by one or more of the following patents: United States: 5,511,185; 5,657,438; 5,701,139; 5,870,559; 5,958,008; 5,974,572; 6,137,782; 6,138,157; 6,144,962; 6,205,122; 6,237,006; 6,341,310; 6,360,332; 6,449,739; 6,470,383; 6,477,483; 6,549,944; 6,560,564; 6,564,342; 6,587,969; 6,631,408; 6,631,411; 6,633,912; 6,694,288; 6,738,813; 6,738,933; 6,754,701; 6,792,460 and 6,810,494. Australia: 763468 and 762554. Other patents pending. All rights reserved.

Mercury, Mercury Interactive, the Mercury logo, the Mercury Interactive logo, LoadRunner, WinRunner, SiteScope and TestDirector are trademarks of Mercury Interactive Corporation and may be registered in certain jurisdictions. The absence of a trademark from this list does not constitute a waiver of Mercury's intellectual property rights concerning that trademark.

All other company, brand and product names may be trademarks or registered trademarks of their respective holders. Mercury disclaims any responsibility for specifying which marks are owned by which companies or which organizations.

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

© 1993 - 2005 Mercury Interactive Corporation, All rights reserved

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

Multi-Volume Chapter Summary

WinRunner user documentation is divided into two volumes:

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

A summary of the various chapters in each guide is provided below:

Mercury WinRunner Basic Features User's Guide

PART I: STARTING THE TESTING PROCESS

Chapter 1: Introduction	3
Chapter 2: WinRunner at a Glance	11

PART II: INTRODUCING THE GUI MAP

Chapter 3: Understanding How WinRunner Identifies GUI Objects ..	25
Chapter 4: Understanding Basic GUI Map Concepts	33
Chapter 5: Working in the Global GUI Map File Mode	45
Chapter 6: Working in the GUI Map File per Test Mode	65
Chapter 7: Editing the GUI Map	71

PART III: CREATING TESTS—BASIC

Chapter 8: Designing Tests	93
Chapter 9: Checking GUI Objects	127
Chapter 10: Working with Web Objects.....	175
Chapter 11: Working with ActiveX and Visual Basic Controls	217
Chapter 12: Checking PowerBuilder Applications.....	237
Chapter 13: Checking Table Contents.....	247
Chapter 14: Checking Databases	259
Chapter 15: Checking Bitmaps	321
Chapter 16: Checking Text	331
Chapter 17: Checking Dates	349
Chapter 18: Creating Data-Driven Tests.....	365
Chapter 19: Synchronizing the Test Run.....	411

PART IV: RUNNING TESTS—BASIC

Chapter 20: Understanding Test Runs.....	427
Chapter 21: Analyzing Test Results	453

PART V: CONFIGURING BASIC SETTINGS

Chapter 22: Setting Properties for a Single Test.....	509
Chapter 23: Setting Global Testing Options	531

Mercury WinRunner Advanced Features User's Guide

PART I: WORKING WITH THE GUI MAP

Chapter 1: Merging GUI Map Files	3
Chapter 2: Configuring the GUI Map	15
Chapter 3: Learning Virtual Objects	37

PART II: CREATING TESTS—ADVANCED

Chapter 4: Defining and Using Recovery Scenarios	45
Chapter 5: Handling Web Exceptions.....	87
Chapter 6: Using Regular Expressions	93

PART III: PROGRAMMING WITH TSL

Chapter 7: Enhancing Your Test Scripts with Programming	103
Chapter 8: Generating Functions.....	121
Chapter 9: Calling Tests	131
Chapter 10: Creating User-Defined Functions.....	147
Chapter 11: Employing User-Defined Functions in Tests	157
Chapter 12: Calling Functions from External Libraries.....	175
Chapter 13: Creating Dialog Boxes for Interactive Input.....	183

PART IV: RUNNING TESTS—ADVANCED

Chapter 14: Running Batch Tests.....	193
Chapter 15: Running Tests from the Command Line.....	201

PART V: DEBUGGING TESTS

Chapter 16: Controlling Your Test Run	225
Chapter 17: Using Breakpoints	231
Chapter 18: Monitoring Variables	241

PART VI: CONFIGURING ADVANCED SETTINGS

Chapter 19: Customizing the Test Script Editor.....251
Chapter 20: Customizing the WinRunner User Interface.....261
Chapter 21: Setting Testing Options from a Test Script.....285
Chapter 22: Customizing the Function Generator.....313
Chapter 23: Initializing Special Configurations.....329

PART VII: WORKING WITH OTHER MERCURY PRODUCTS

Chapter 24: Working with Business Process Testing.....335
Chapter 25: Integrating with QuickTest Professional371
Chapter 26: Managing the Testing Process379
Chapter 27: Testing Systems Under Load415

Table of Contents

Multi-Volume Chapter Summary	iii
Mercury WinRunner Basic Features User's Guide.....	iii
Mercury WinRunner Advanced Features User's Guide.....	iv
Welcome to Mercury WinRunner	xv
Using this Guide.....	xv
WinRunner Documentation Set.....	xvi
Online Resources	xvii
Documentation Updates	xviii
Typographical Conventions.....	xix

PART I: STARTING THE TESTING PROCESS

Chapter 1: Introduction	3
WinRunner Testing Modes.....	4
The WinRunner Testing Process	5
Sample Application	8
Integrating with Other Mercury Interactive Products	9
Chapter 2: WinRunner at a Glance	11
Starting WinRunner	11
The Main WinRunner Window	14
The Test Editor Window	16
Using WinRunner Commands.....	17
Loading WinRunner Add-Ins	20

PART II: INTRODUCING THE GUI MAP

Chapter 3: Understanding How WinRunner Identifies GUI Objects ..	25
About Identifying GUI Objects	25
How a Test Identifies GUI Objects	27
Logical Names	29
The GUI Map	30
Setting the Window Context	31
Chapter 4: Understanding Basic GUI Map Concepts	33
About the GUI Map.....	33
Viewing GUI Object Properties	34
Teaching WinRunner the GUI of Your Application	40
Finding an Object or Window in the GUI Map.....	41
General Guidelines for Working with GUI Map Files	41
Deciding Which GUI Map File Mode to Use	42
Chapter 5: Working in the Global GUI Map File Mode	45
About the Global GUI Map File Mode	45
Sharing a GUI Map File among Tests.....	47
Teaching WinRunner the GUI of Your Application	48
Saving the GUI Map	57
Loading the GUI Map File.....	59
Guidelines for Working in the Global GUI Map File Mode	63
Chapter 6: Working in the GUI Map File per Test Mode	65
About the GUI Map File per Test Mode	65
Specifying the GUI Map File per Test Mode	67
Working in the GUI Map File per Test Mode	68
Guidelines for Working in the GUI Map File per Test Mode	69

Chapter 7: Editing the GUI Map	71
About Editing the GUI Map	72
The GUI Map Editor	73
The Run Wizard.....	75
Modifying Logical Names and Physical Descriptions.....	77
How WinRunner Handles Varying Window Labels	79
Using Regular Expressions in the Physical Description	81
Copying and Moving Objects between Files.....	82
Finding an Object in a GUI Map File	84
Finding an Object in Multiple GUI Map Files	85
Manually Adding an Object to a GUI Map File	86
Deleting an Object from a GUI Map File	86
Clearing a GUI Map File.....	87
Filtering Displayed Objects	88
Saving Changes to the GUI Map.....	89

PART III: CREATING TESTS—BASIC

Chapter 8: Designing Tests	93
About Creating Tests	94
Understanding the WinRunner Test Window	95
Planning a Test.....	96
Creating Tests Using Context Sensitive Recording.....	97
Creating Tests Using Analog Recording.....	103
Guidelines for Recording a Test	105
Adding Checkpoints to Your Test.....	107
Working with Data-Driven Tests.....	107
Adding Synchronization Points to a Test	108
Measuring Transactions	108
Activating Test Creation Commands Using Softkeys	111
Programming a Test	114
Editing a Test.....	114
Managing Test Files	116

Chapter 9: Checking GUI Objects	127
About Checking GUI Objects.....	128
Checking a Single Property Value.....	130
Checking a Single Object	132
Checking Two or More Objects in a Window	135
Checking All Objects in a Window.....	137
Understanding GUI Checkpoint Statements	140
Using an Existing GUI Checklist in a GUI Checkpoint.....	141
Modifying GUI Checklists.....	143
Understanding the GUI Checkpoint Dialog Boxes.....	148
Property Checks and Default Checks.....	158
Specifying Arguments for Property Checks	164
Editing the Expected Value of a Property	170
Modifying the Expected Results of a GUI Checkpoint.....	172
Chapter 10: Working with Web Objects	175
About Working with Web Objects.....	176
Viewing Recorded Web Object Properties	176
Using Web Object Properties in Your Tests	178
Checking Web Objects.....	187
Chapter 11: Working with ActiveX and Visual Basic Controls	217
About Working with ActiveX and Visual Basic Controls	218
Choosing Appropriate Support for Visual Basic Applications	222
Viewing ActiveX and Visual Basic Control Properties.....	223
Retrieving and Setting the Values of ActiveX and Visual Basic Control Properties	226
Activating an ActiveX Control Method.....	229
Working with Visual Basic Label Controls	230
Checking Sub-Objects of ActiveX and Visual Basic Controls.....	232
Using TSL Table Functions with ActiveX Controls.....	235
Chapter 12: Checking PowerBuilder Applications	237
About Checking PowerBuilder Applications	238
Checking Properties of DropDown Objects.....	238
Checking Properties of DataWindows	241
Checking Properties of Objects within DataWindows	243
Working with Computed Columns in DataWindows.....	245
Chapter 13: Checking Table Contents	247
About Checking Table Contents.....	247
Checking Table Contents with Default Checks.....	249
Checking Table Contents while Specifying Checks	250
Understanding the Edit Check Dialog Box.....	253

Chapter 14: Checking Databases	259
About Checking Databases.....	260
Creating a Runtime Database Record Checkpoint.....	264
Editing a Runtime Database Record Checklist.....	272
Creating a Default Check on a Database	277
Creating a Custom Check on a Database.....	280
Messages in the Database Checkpoint Dialog Boxes	283
Working with the Database Checkpoint Wizard.....	283
Understanding the Edit Check Dialog Box.....	291
Modifying a Standard Database Checkpoint	297
Modifying the Expected Results of a Standard Database Checkpoint	308
Parameterizing Standard Database Checkpoints	310
Specifying a Database.....	314
Using TSL Functions to Work with a Database.....	316
Chapter 15: Checking Bitmaps	321
About Checking Bitmaps.....	321
Creating Bitmap Checkpoints.....	323
Checking Window and Object Bitmaps.....	326
Checking Area Bitmaps	328
Chapter 16: Checking Text	331
About Checking Text.....	331
Reading Text.....	333
Searching for Text	337
Comparing Text	342
Teaching Fonts to WinRunner	342
Chapter 17: Checking Dates	349
About Checking Dates.....	349
Testing Date Operations.....	350
Testing Two-Character Date Applications	351
Setting Date Formats	352
Using an Existing Date Format Configuration File.....	354
Checking Dates in GUI Objects	355
Checking Dates with TSL	357
Overriding Date Settings	358

Chapter 18: Creating Data-Driven Tests	365
About Creating Data-Driven Tests	366
The Data-Driven Testing Process.....	366
Creating a Basic Test for Conversion	367
Converting a Test to a Data-Driven Test.....	369
Preparing the Data Table.....	383
Importing Data from a Database.....	390
Running and Analyzing Data-Driven Tests	395
Assigning the Main Data Table for a Test	396
Using Data-Driven Checkpoints and Bitmap Synchronization Points	397
Using TSL Functions with Data-Driven Tests	402
Guidelines for Creating a Data-Driven Test.....	409
Chapter 19: Synchronizing the Test Run	411
About Synchronizing the Test Run	411
Waiting for Objects and Windows.....	413
Waiting for Property Values of Objects and Windows.....	414
Waiting for Bitmaps of Objects and Windows	419
Waiting for Bitmaps of Screen Areas.....	421
Tips for Synchronizing Tests	423

PART IV: RUNNING TESTS—BASIC

Chapter 20: Understanding Test Runs	427
About Understanding Test Runs	428
WinRunner Test Run Modes	429
WinRunner Run Commands	433
Choosing Run Commands Using Softkeys	435
Running a Test to Check Your Application	436
Running a Test to Debug Your Test Script	437
Running a Test to Update Expected Results.....	438
Running a Test to Check Date Operations	442
Supplying Values for Input Parameters When Running a Test	447
Controlling the Test Run with Testing Options	448
Solving Common Test Run Problems	449

Chapter 21: Analyzing Test Results	453
About Analyzing Test Results	454
Understanding the Unified Report View Results Window	456
Customizing the Test Results Display	466
Understanding the WinRunner Report View Results Window	467
Viewing the Results of a Test Run	474
Viewing Checkpoint Results	481
Analyzing the Results of a Single-Property Check	483
Analyzing the Results of a GUI Checkpoint	484
Analyzing the Results of a GUI Checkpoint on Table Contents	486
Analyzing the Expected Results of a GUI Checkpoint on Table Contents	489
Analyzing the Results of a Bitmap Checkpoint	493
Analyzing the Results of a Database Checkpoint	494
Analyzing the Expected Results of a Content Check in a Database Checkpoint	496
Updating the Expected Results of a Checkpoint in the WinRunner Report View.....	499
Viewing the Results of a File Comparison	500
Viewing the Results of a GUI Checkpoint on a Date.....	502
Reporting Defects Detected During a Test Run.....	503

PART V: CONFIGURING BASIC SETTINGS

Chapter 22: Setting Properties for a Single Test	509
About Setting Properties for a Single Test	509
Setting Test Properties from the Test Properties Dialog Box	510
Documenting General Test Information	512
Documenting Descriptive Test Information	514
Managing Test Parameters	515
Associating Add-ins with a Test	519
Reviewing Current Test Settings	521
Defining Startup Applications and Functions	523

Chapter 23: Setting Global Testing Options	531
About Setting Global Testing Options	531
Setting Global Testing Options from the General Options Dialog Box	532
Setting General Options	535
Setting Folder Options	541
Setting Recording Options	545
Setting Test Run Options	562
Setting Notification Options	579
Setting Appearance Options	585
Choosing Appropriate Timeout and Delay Settings	589
Index	593

Welcome to Mercury WinRunner

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

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

Using this Guide

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

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

This guide contains the following parts:

Part I Starting the Testing Process

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

Part II Introducing 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—Basic

Describes how to create test scripts, insert checkpoints, and assign parameters.

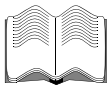
Part IV Running Tests—Basic

Describes how to run tests from within WinRunner and analyze test results.

Part V Configuring Basic Settings

Describes how to change WinRunner's default settings, both per test and globally.

WinRunner Documentation Set



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

WinRunner Advanced Features User's Guide provides information on the more advanced WinRunner features you can use to meet the special testing requirements of your application.

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

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

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

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

Online Resources

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

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

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

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

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

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

Sample Tests includes utilities and sample tests with accompanying explanations.

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

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

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

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

Documentation Updates

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

To download updated documentation:

- 1** In the Customer Support Web site, click the **Documentation** link.
- 2** Under **Select Product Name**, select **WinRunner**.
Note that if **WinRunner** does not appear in the list, you must add it to your customer profile. Click **My Account** to update your profile.
- 3** Click **Retrieve**. The Documentation page opens and lists the documentation available for the current release and for previous releases. If a document was recently updated, **Updated** appears next to the document name.
- 4** Click a document link to download the documentation.

Typographical Conventions

This book uses the following typographical conventions:

1, 2, 3	Bold numbers indicate steps in a procedure.
>	The greater-than sign separates menu levels (for example, File > Open).
Stone Sans	The Stone Sans font indicates names of interface elements (for example, the Run button) and other items that require emphasis.
Bold	Bold text indicates method or function names.
<i>Italics</i>	<i>Italic</i> text indicates method or function arguments, file names in syntax descriptions, and book titles. It is also used when introducing a new term.
<>	Angle brackets enclose a part of a file path or URL address that may vary from user to user (for example, < MyProduct installation folder >\bin).
Arial	The Arial font is used for examples and text that is to be typed literally.
Arial bold	The Arial bold font is used in syntax descriptions for text that should be typed literally.
SMALL CAPS	The SMALL CAPS font indicates keyboard keys.
...	In a line of syntax, an ellipsis indicates that more items of the same format may be included. In a programming example, an ellipsis is used to indicate lines of a program that were intentionally omitted.
[]	Square brackets enclose optional arguments.
	A vertical bar indicates that one of the options separated by the bar should be selected.

Welcome

Part I

Starting the Testing Process

1

Introduction

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.

Mercury WinRunner is the powerful test automation solution for the enterprise. It 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 release software of superior quality.

You can also convert existing WinRunner tests to scripted components, or create new scripted components. Scripted components are part of Business Process Testing in Mercury Quality Center, which utilizes a keyword-driven methodology for testing applications. Scripted components are reusable modular scripts that can be created in WinRunner, and then used in business process tests.

The information, examples, and screen captures in this guide focus specifically on working with WinRunner tests. Much of the information that is relevant for tests is also relevant for scripted components, which have functionality that is similar to tests.

Integration with Quality Center and how to work with scripted components is described in the *Mercury WinRunner Advanced Features User's Guide*. For more information, refer to the *Business Process Testing User's Guide*.

WinRunner Testing Modes

WinRunner facilitates test creation by recording how you work on your application. As you point and click GUI (Graphical User Interface) objects in your application, WinRunner automatically generates a test script in its 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 and Analog.

Context Sensitive

Context Sensitive mode records your actions on the application you are testing 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 you are testing, 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 file. The GUI map files are maintained separately from your test scripts and the same GUI map file (or files) can be used for multiple tests. 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 reuse your Context Sensitive test scripts on future versions of your application.

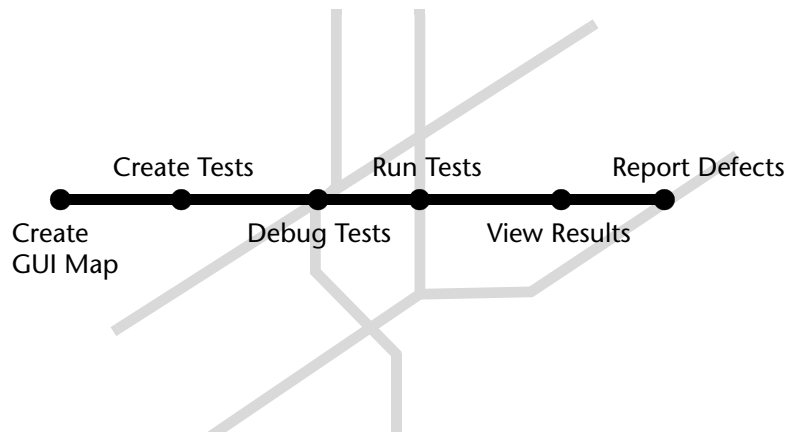
To run a test, you play back the test script. WinRunner emulates a user by moving the mouse pointer over your application, selecting objects, and entering keyboard input. WinRunner reads the object descriptions in the GUI map and then searches in the application you are testing for objects matching these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

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 you are testing. 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 you are testing.

Debug Tests

You run tests in Debug mode to check whether 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 you are testing to the expected data captured earlier. If any mismatches are found, WinRunner captures them as *actual results*.

Note: Verify mode is only relevant when running tests, not components. When working with components, the application is verified when the component is run as part of a business process test in Quality Center.

View Results

You view results to determine the success or failure of your 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 you are testing, you can report information about the defect directly from the Test Results window. This information is managed by the quality assurance manager, who tracks the defect until it is fixed.

You can also insert `qcdb_add_defect` statements to your test script that instruct WinRunner to add a defect to a Quality Center project based on conditions you define in your test script.

Sample Application

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 A and Flight B are installed in addition to the regular Windows-based 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. 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.

For more information about integration with QuickTest Professional, Quality Center, Business Process Testing, and LoadRunner, refer to the *Mercury WinRunner Advanced Features User's Guide*.

Mercury 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.

Mercury Quality Center

Quality Center (formerly TestDirector) is an application quality management product. It helps quality assurance personnel plan and organize the testing process. With Quality Center you can create a database of scripted components and 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 and scripted components directly to your Quality Center database. You can also run tests in WinRunner and then use Quality Center to review the overall results of a testing cycle.

Integrating WinRunner and Quality Center with Business Process Testing support enables you to leverage your investment in existing WinRunner script libraries and improve the test automation process by using the Business Process Testing framework.

Mercury LoadRunner

LoadRunner is the Mercury solution for automated performance testing. 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 you are testing. 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 Editor Window
- ▶ Using WinRunner Commands
- ▶ Loading WinRunner Add-Ins

Starting WinRunner

To start WinRunner for the first time:



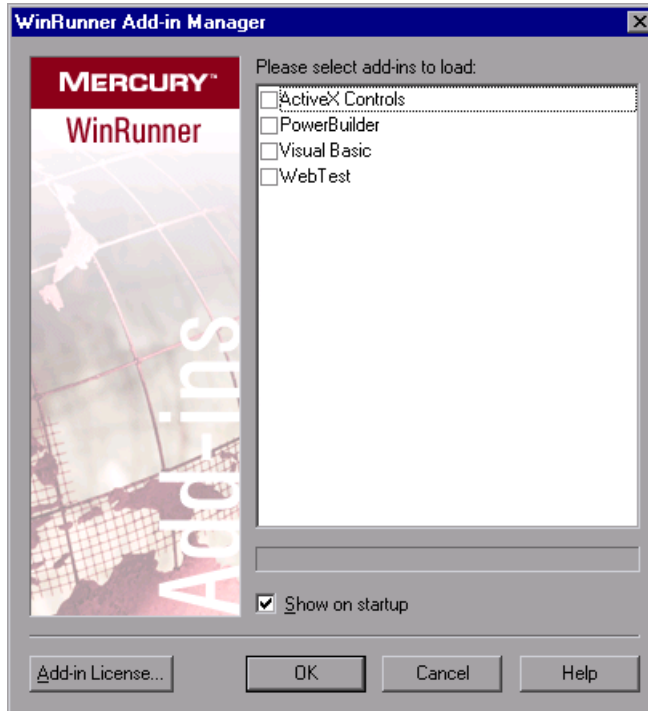
- 1** Choose **Programs > WinRunner > WinRunner** on the **Start** menu.

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 you are testing.

- 2** By default, the WinRunner Add-in Manager dialog box opens.

The WinRunner Add-in Manager dialog box contains a list of the add-ins available on your computer. 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 add-ins that were loaded in the previous WinRunner session are automatically loaded. A progress bar displays how much time is left before the window closes.



Note: The first time you start a new version of WinRunner on your computer, “What’s New in WinRunner” Help also opens.

For more information on the Add-in Manager, see “Loading WinRunner Add-Ins” on page 20.

- 3 The Welcome to WinRunner window opens. From the Welcome to WinRunner window you can click **Create a New Test** to open a new, blank test, click **Open an Existing Test** to select a saved test to open, or click **View a Quick Preview of WinRunner** to view an overview presentation of WinRunner in your default browser.



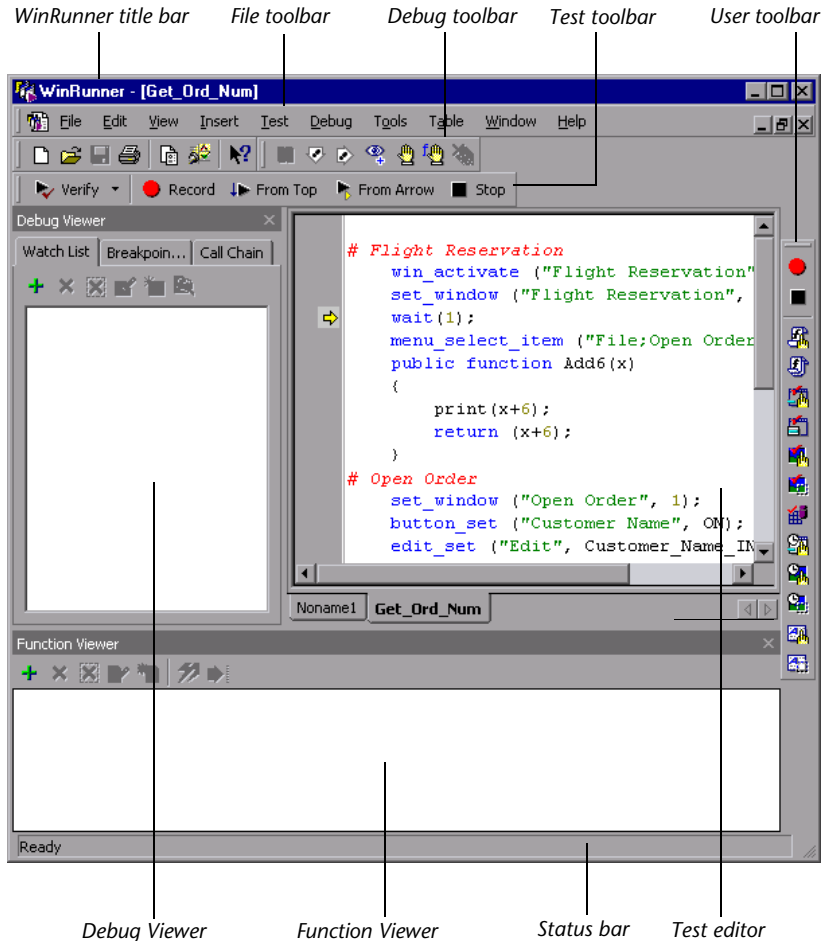
Tip: If you do not want the Welcome to WinRunner window to open the next time you start WinRunner, clear the **Show on startup** check box. Once you have selected this option, you can instruct WinRunner to display the Welcome screen at a later time. Choose **Tools > General Options**, select the **General > 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 access to frequently performed tasks, such as opening and saving tests, and viewing test results.
- ▶ **Debug toolbar**—provides access to options used while debugging tests.
- ▶ **Test toolbar**—provides access to options 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 User 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.
- ▶ **Test Editor**—displays the test script.
- ▶ **Debug Viewer**—displays data from the currently selected debug option: **Watch List**, **Breakpoints**, or **Call Chain**. You can close this pane by clearing all selected debug toggle options in the **Debug** menu.
- ▶ **Function Viewer**—displays loaded functions that you can call from your tests. You can close this pane by clearing the **Function Viewer** toggle option in the **Tools** menu.

An example of the main WinRunner window is shown below:

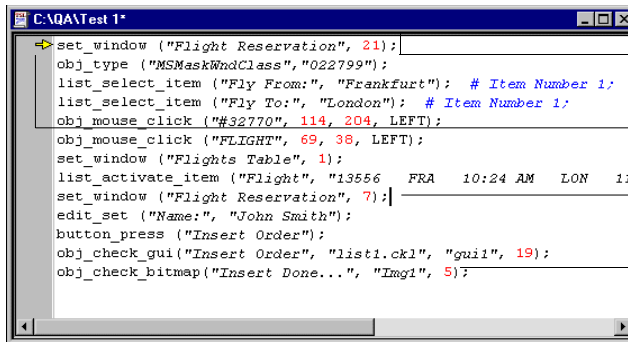


Tip: By default, each test is displayed in a separate tab in the test editor. If there are more test tabs than can fit across the bottom of the editor, you can click the left or right arrow buttons to scroll the test tabs to the left or the right. If tabs are not displayed, you can display them by selecting the **Display test tabs** option in the Appearance category of the General Options dialog box.

The Test Editor Window

You create and run WinRunner tests in the test editor 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 manually entered by programming in Test Script Language (TSL)
- *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



```
C:\QA\Test 1*
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);
```

Test window title bar

Execution arrow

Insertion point

Test script

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

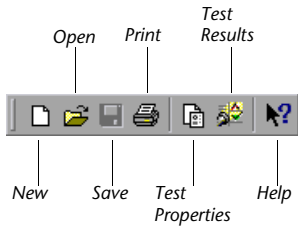
You can change any toolbar to a floating toolbar. In addition, when the User toolbar is a floating toolbar, you can minimize WinRunner and still maintain access to the commands on the User toolbar, so you can work freely with the application you are testing.

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.

The following buttons appear on the File toolbar:

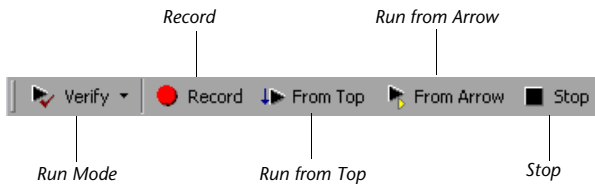


For more information about the File toolbar, see Chapter 8, “Designing Tests.”

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.

The following buttons appear on the Test toolbar:

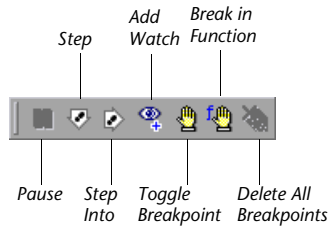


For more information about the Test toolbar, see Chapter 20, “Understanding Test Runs.”

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.

The following buttons appear on the Debug toolbar:

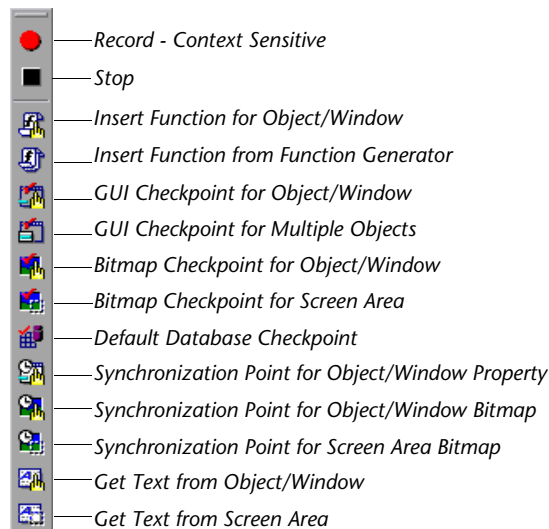


For more information about the Debug toolbar, see Chapter 20, “Understanding Test Runs.”

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.

The User toolbar is a customizable toolbar. You can add or remove buttons to facilitate access to commands commonly used for an application you are testing. The following buttons appear by default on the User toolbar:



For information on customizing the User toolbar, refer to Chapter 20, “Customizing the WinRunner User Interface” in the *Mercury WinRunner Advanced Features User’s Guide*.

Activating Commands Using Softkeys

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

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

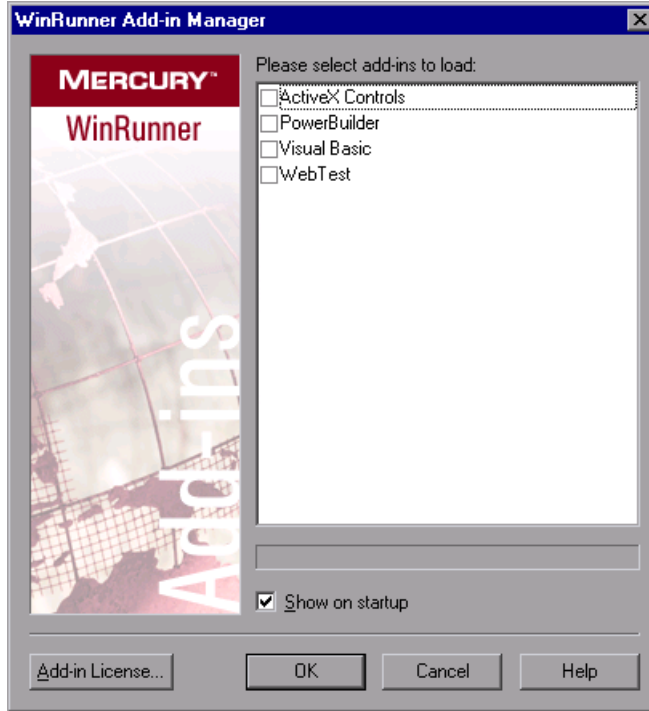
For a list of default WinRunner softkey configurations and information about redefining WinRunner softkeys, refer to Chapter 20, “Customizing the WinRunner User Interface” in the *Mercury WinRunner Advanced Features User’s Guide*.

Loading WinRunner Add-Ins

If you installed core add-ins such as the WebTest, Visual Basic, PowerBuilder, or ActiveX Controls while installing WinRunner or afterward, or if you have installed external add-ins, you can specify which add-ins to load at the beginning of each WinRunner session.

By default, the **Add-In Manager** dialog box opens when you start WinRunner. 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 starts, 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 and load the selected add-ins.

The Add-in Manager displays the list of add-ins available from your computer. The core 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 with a seat license, 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. For more information, refer to the *WinRunner Installation Guide*.

You can determine whether to display the **Add-In Manager** dialog box each time WinRunner opens 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 23, “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, refer to Chapter 15, “Running Tests from the Command Line” in the *Mercury WinRunner Advanced Features User’s Guide*.

Part II

Introducing 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
- ▶ 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, refer to Chapter 1, “Merging GUI Map Files” in the *Mercury WinRunner Advanced Features User’s Guide*.

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, refer to Chapter 2, “Configuring the GUI Map” in the *Mercury WinRunner Advanced Features User’s Guide*.

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, refer to Chapter 3, “Learning Virtual Objects” in the *Mercury WinRunner Advanced Features User’s Guide*.

How a Test Identifies GUI Objects

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

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:

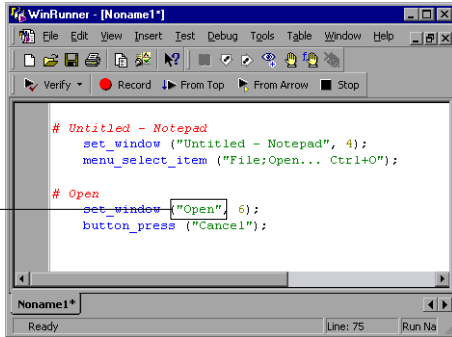
```
{property1:value1, property2:value2, property3:value3, ...}
```

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}
```

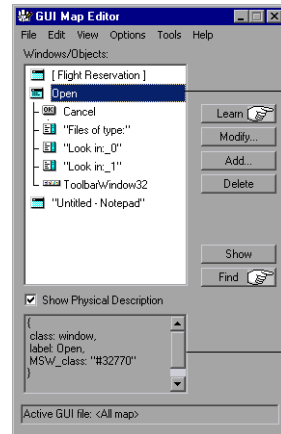
The class property indicates the object's type. Each object belongs to a different class, according to its functionality: window, push button, list, radio button, menu, etc.

Test Script



logical name

GUI Map



logical name

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

Application Being Tested



- 3 WinRunner uses the physical description to find an object in the application

"Open" window label

Each class has a set of default properties that WinRunner learns. For a detailed description of all properties, refer to Chapter 2, "Configuring the GUI Map" in the *Mercury WinRunner Advanced Features User's Guide*.

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 31.

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.

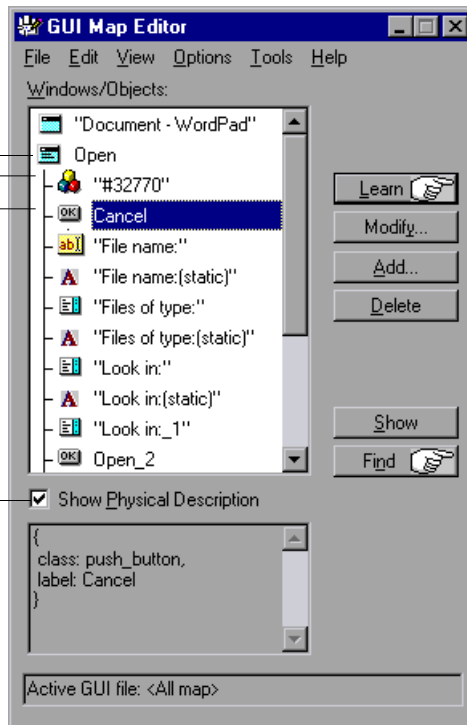
In the GUI Map Editor, you can view either the contents of the entire GUI map or the contents of individual GUI map files. GUI objects are grouped according to the window in which they appear in the application.

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.

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

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 42.

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.

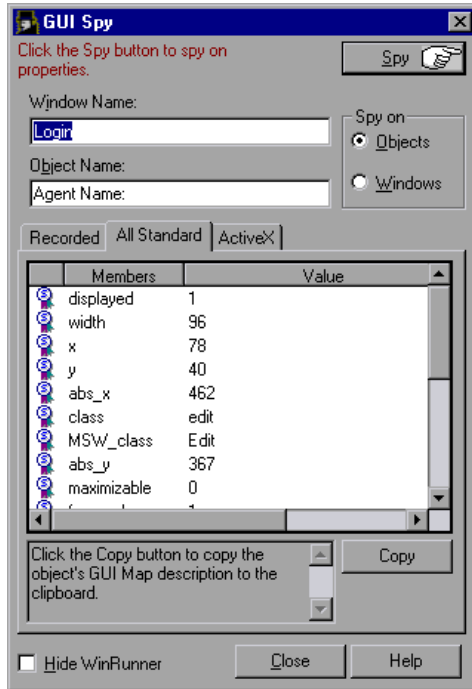
Viewing GUI Object Properties

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:

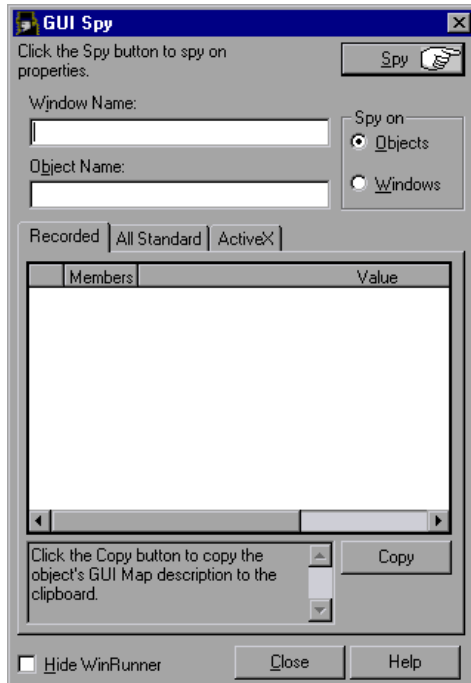


Tip: You can resize the GUI Spy to view the entire contents at once.

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

To spy on a GUI object or window:

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



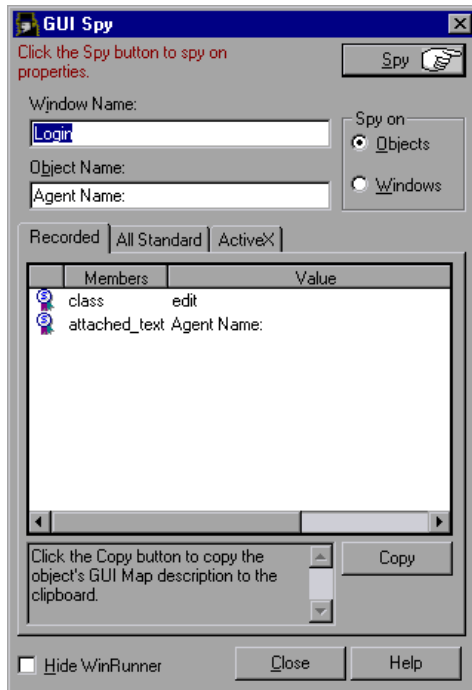
By default, the GUI Spy displays the Recorded tab, which enables you to view the properties of standard GUI objects that WinRunner records or learns.

- 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 In the Spy on box, select **Objects** or **Windows**.
 - 3 Select **Hide WinRunner** if you want to hide the WinRunner screen (but not the GUI Spy) while you spy on objects.

- 4 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:



- 5 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.

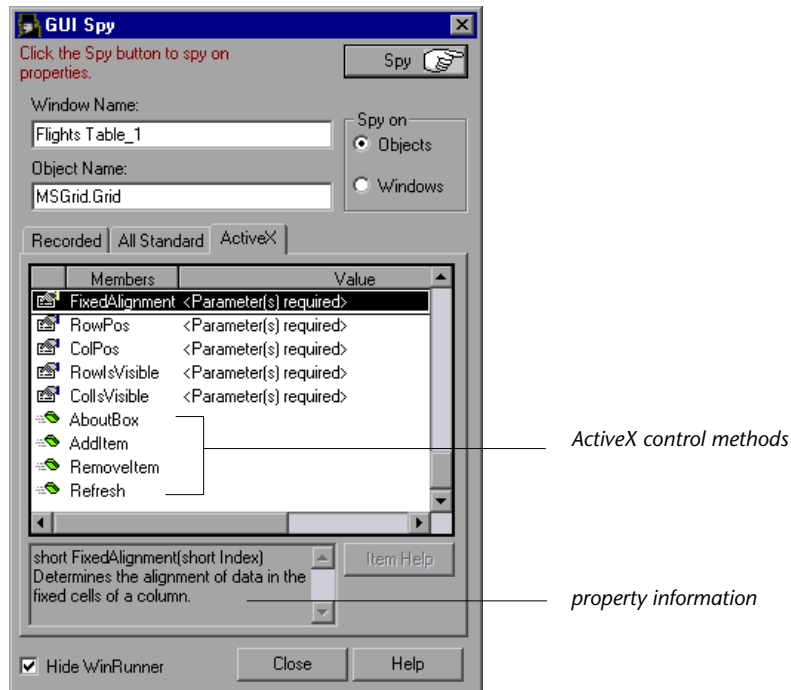
Clicking Copy in the previous example pastes the following physical description to the Clipboard:

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

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



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.

6 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. WinRunner can learn this information in the following ways:

- ▶ 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
- ▶ using the RapidTest Script wizard to learn the properties of all GUI objects in every window in your application

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 the add-in documentation.

When you work in the *Global GUI Map File* mode, you must first take some administrative steps in addition to utilizing one of the three ways mentioned above. For example, you must save the object in a permanent GUI file, and make sure the file is loaded when the test is running. For more information, see Chapter 5, “Working in the Global GUI Map File Mode.” However, in the *Gui File Per Test* mode you do not need to take any extra steps. WinRunner performs the administrative tasks automatically.

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” and Chapter 6, “Working in the GUI Map File per Test 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.

- ▶ 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 88.

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.

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.

	GUI Map File per Test	Global GUI Map File
Advantages	<ol style="list-style-type: none"> 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. 	<ol style="list-style-type: none"> 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.
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.
Suggested Method	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 77.

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 63.

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.”

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.

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.

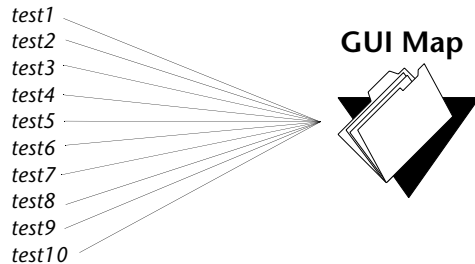
Similarly, since GUI map files are independent of tests, they are not automatically loaded when you open a test. Therefore, you must load the appropriate GUI map files before you run tests. WinRunner uses these files to help locate the objects in the application being tested. It is most efficient to 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, refer to Chapter 23, “Initializing Special Configurations” in the *Mercury WinRunner Advanced Features User’s Guide*.

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.

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 2, “Configuring the GUI Map” in the *Mercury WinRunner Advanced Features User’s Guide*.

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 gives WinRunner a 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 the add-in documentation.

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, refer to Chapter 23, “Initializing Special Configurations” in the *Mercury WinRunner Advanced Features User’s Guide*.

To teach WinRunner your application using the RapidTest Script wizard:

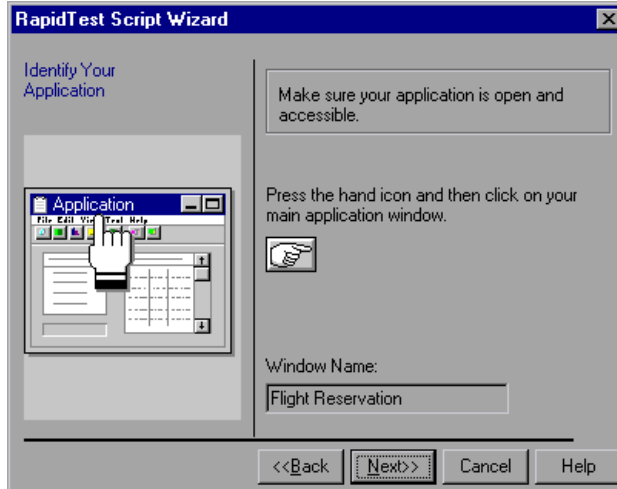
- 1 Choose **Insert > RapidTest Script Wizard**. The RapidTest Script wizard welcome screen opens.



Click **Next**.

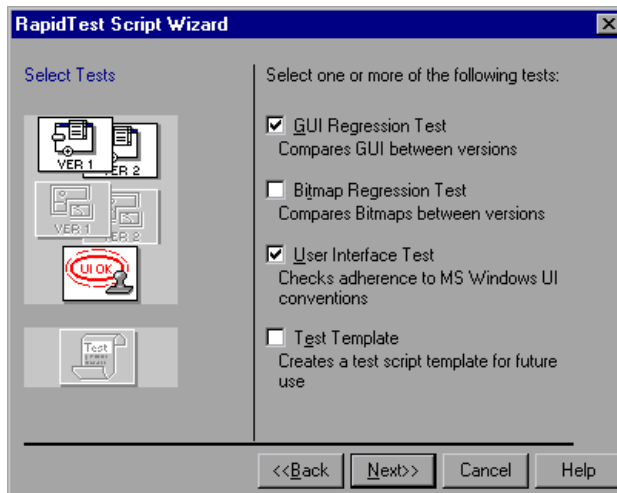
Note: The **RapidTest Script Wizard** option is not available when you use the WinRunner run-only version, when you work in *GUI file per test* mode, or when you load the WebTest add-in or certain other add-ins. Refer to the add-in documentation to see whether the RapidTest Script wizard is available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

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



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.



- 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

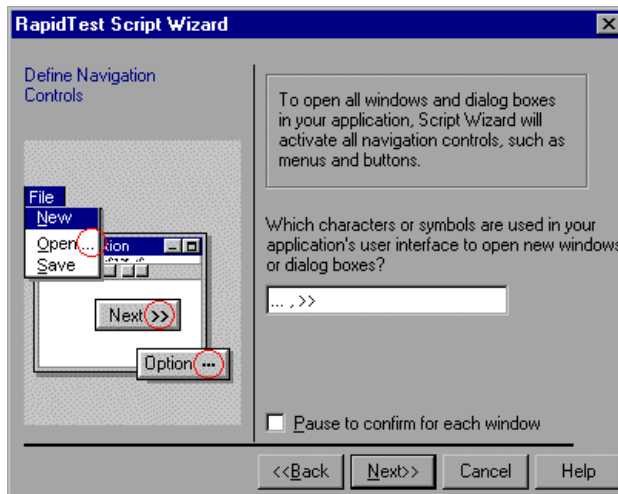
When you run this test, WinRunner searches the user interface of your application and reports each case that does not adhere to Microsoft Windows standards.

- **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.



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 The **Set the Learning Flow** screen opens.

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 The **Start Application** screen opens.

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 The **Save Files** screen opens.

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 The **Congratulations** screen opens.

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. For information on recording in Context Sensitive mode, see Chapter 8, “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 57.

Tip: If you do not want WinRunner to add information 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 23, “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.

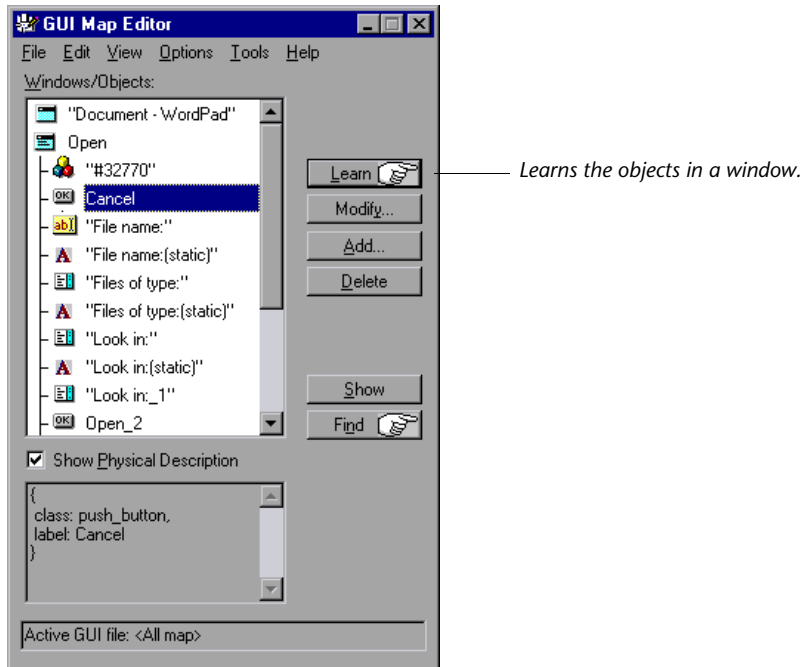
Teaching WinRunner the GUI Using the GUI Map Editor

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.

- 2 Click **Learn**. The mouse pointer becomes a pointing hand. Place the pointing hand on the object to learn, and click the left mouse button.



- 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.)

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.

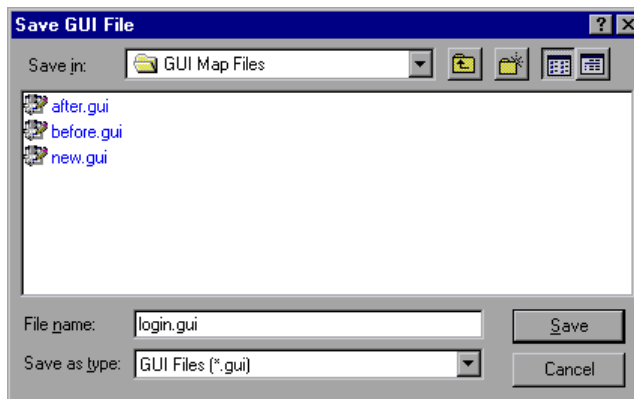
Saving the GUI Map

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.



- 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.

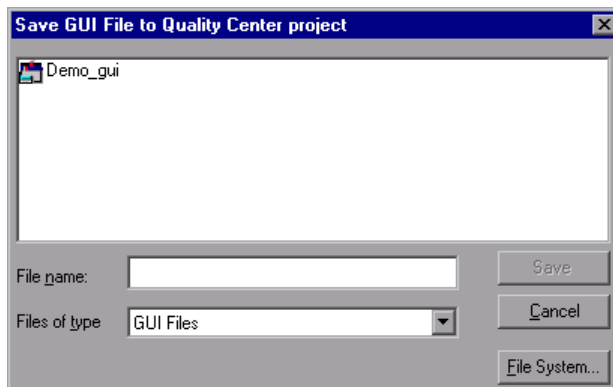
You can also move objects from the temporary file to an existing GUI map file. For details, see “Copying and Moving Objects between Files” on page 82.

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

Note: You can only save GUI map files to a Quality Center database if you are working with Quality Center. For additional information, refer to Chapter 26, “Managing the Testing Process” in the *Mercury WinRunner Advanced Features User’s Guide*.

- 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 Quality Center project dialog box opens.



- 5 In the **File name** text box, enter a name for the GUI map file. Use a descriptive name that will help you easily identify it later.
- 6 Click **Save** to save the GUI map file to a Quality Center 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, refer to Chapter 23, “Initializing Special Configurations” in the *Mercury WinRunner Advanced Features User’s Guide*.

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");
```

Refer to Chapter 8, “Generating Functions,” in the *Mercury WinRunner Advanced Features User’s Guide* for information on how to use the Function Generator.

- 3** Run the test to load the file. See Chapter 20, “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, refer to Chapter 7, “Enhancing Your Test Scripts with Programming” in the *Mercury WinRunner Advanced Features User’s Guide*. 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 Quality Center database, using the GUI Map Editor.

Note: You can only load GUI map files from a Quality Center database if you are connected to a Quality Center project. For additional information, refer to Chapter 26, “Managing the Testing Process” in the *Mercury WinRunner Advanced Features User’s Guide*.

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.



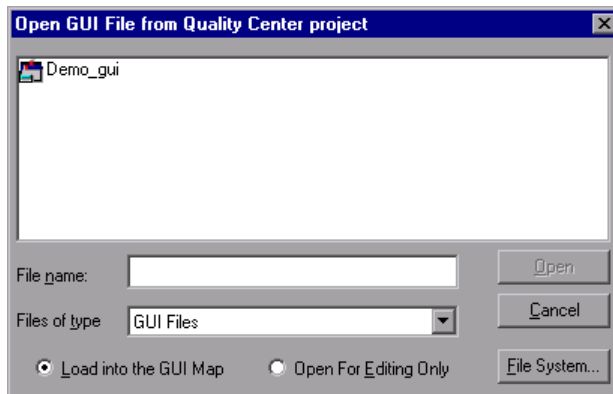
Note that by default, the file is loaded into the GUI map. If you only want to edit the GUI map file, click **Open for Editing Only**. See Chapter 7, “Editing the GUI Map,” for information about editing the GUI map file.

- 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 Quality Center 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 Quality Center 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 77.
- ▶ 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 23, “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 using 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 41.

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.

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 23, “Setting Global Testing Options.”
- 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, refer to Chapter 11, “Employing User-Defined Functions in Tests” in the *Mercury WinRunner Advanced Features User’s Guide*.
- 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, it is recommended that you 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, refer to Chapter 1, “Merging GUI Map Files” in the *Mercury WinRunner Advanced Features User’s Guide*.

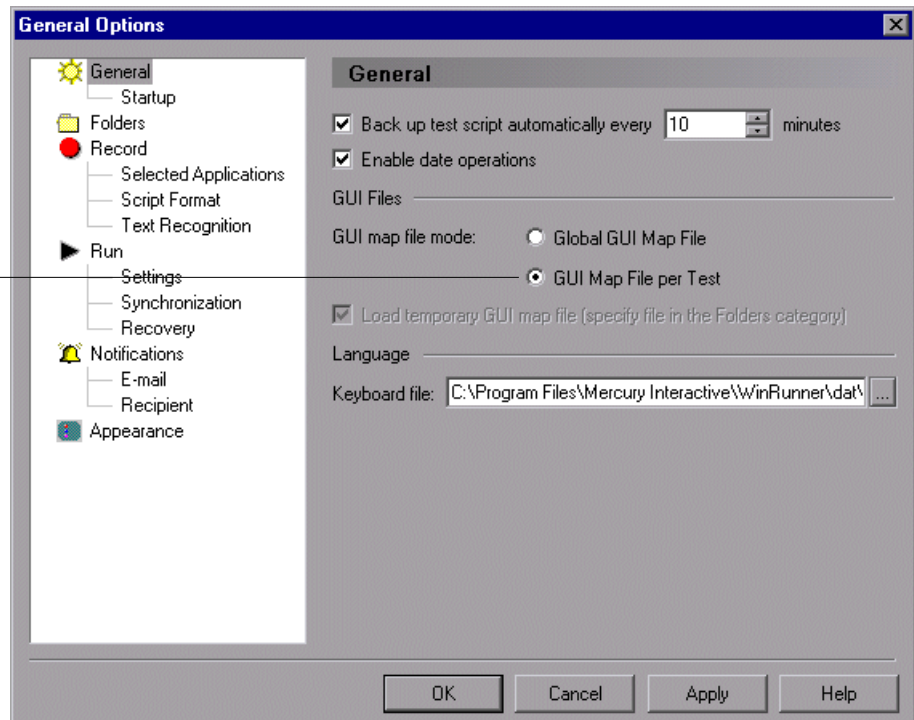
Specifying the GUI Map File per Test Mode

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.



- 4** Click **OK** to close the dialog box.
- 5** A dialog box opens warning you that changes will not take effect until you close and restart WinRunner. Click **OK**.

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

- 6 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 23, “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 either by recording, or by using the Learn feature. 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 77.

- 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 insert any **GUI_load** statements into your tests.
- ▶ 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.
- ▶ Do not call other tests that utilize the Global GUI Map mode.

For additional guidelines for working with GUI maps, see “General Guidelines for Working with GUI Map Files” on page 41.

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

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:

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

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.

While working with the GUI Map Editor, you can:

- ▶ manually edit the GUI map
- ▶ 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.

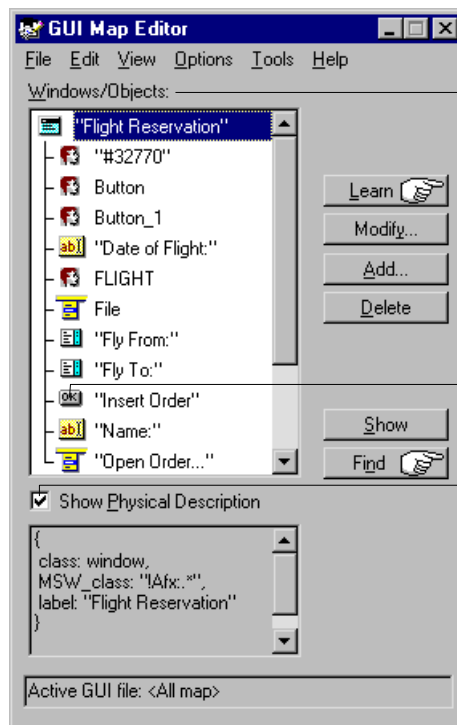
Note: If you are working in the *GUI Map File per Test* mode, you should not manually load or unload GUI map files.

The 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

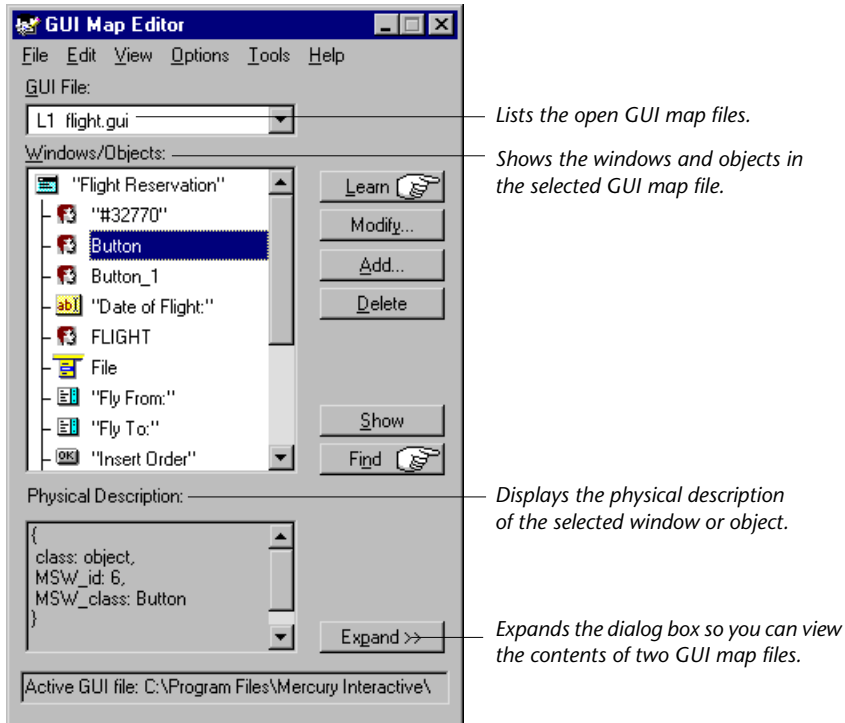


Displays all windows and objects in the GUI map.

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**.



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  
}
```

Notes:

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.

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.

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.)

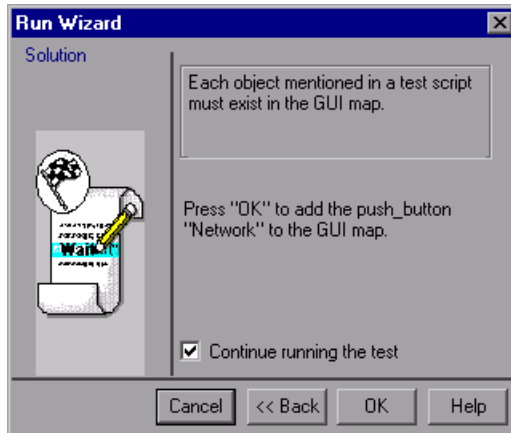
For example, suppose you run a test in which you click the Network button in an Open window in your application. This portion of your script may appear as follows:

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

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



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



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

In some cases, the Run wizard edits the test script, 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.

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:

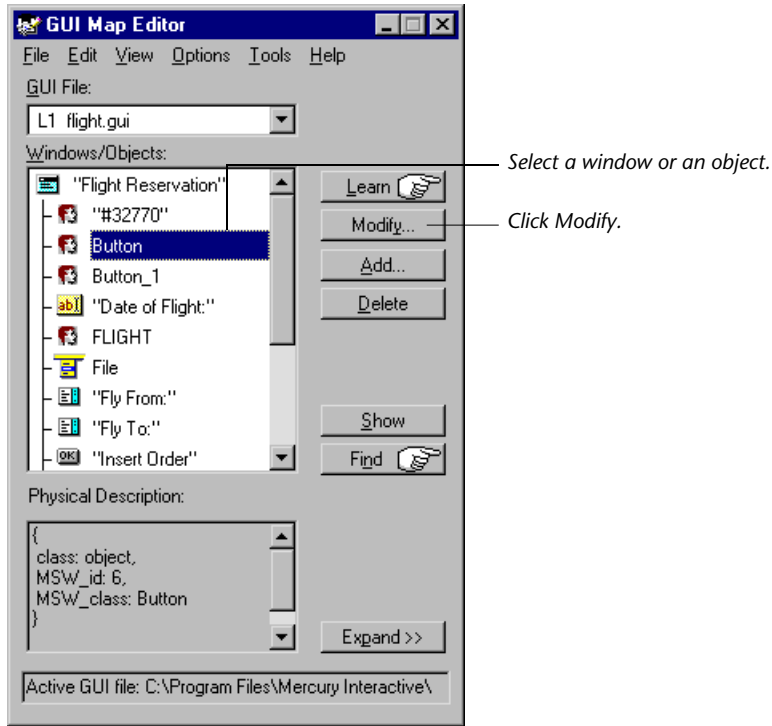
```
Insert button:{class:push_button, label:Add}
```

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

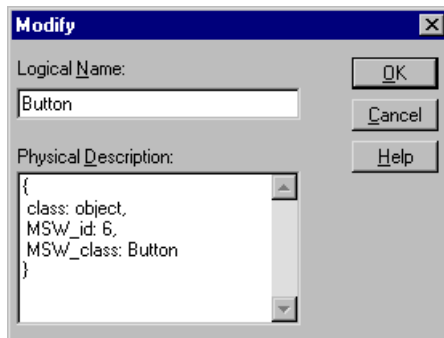
To modify an object’s logical name or physical description in a GUI map file:

- 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.



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



- 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

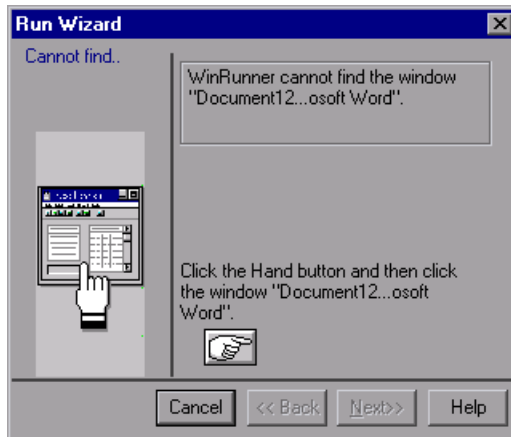
Windows often have varying labels. For example, the main window in a text application may display the file name and application name in the title bar.

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.

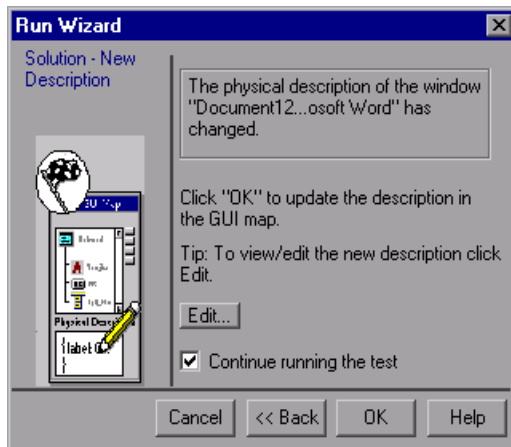
For example, suppose you record a test on the main window of Microsoft Word. WinRunner learns the following physical description:

```
{
  class: window,
  label: "Microsoft Word - Document11",
  MSW_class: OpusApp
}
```

Suppose you run your test when Document 12 is open in Microsoft Word. When WinRunner cannot find the window, the Run wizard opens:



You click the Hand button and click the appropriate Microsoft Word window, so that WinRunner will learn it. You are prompted to instruct WinRunner to update the window's description in the GUI map.



If you click Edit, you can see that WinRunner has modified the window's physical description to include regular expressions:

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

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

These regular expressions enable WinRunner to recognize the window regardless of the name appearing after the Microsoft Word - 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. For more information, refer to Chapter 2, “Configuring the GUI Map” in the *Mercury WinRunner Advanced Features User’s Guide*.

For more information about regular expressions, refer to Chapter 6, “Using Regular Expressions” in the *Mercury WinRunner Advanced Features User’s Guide*.

Copying and Moving Objects between Files

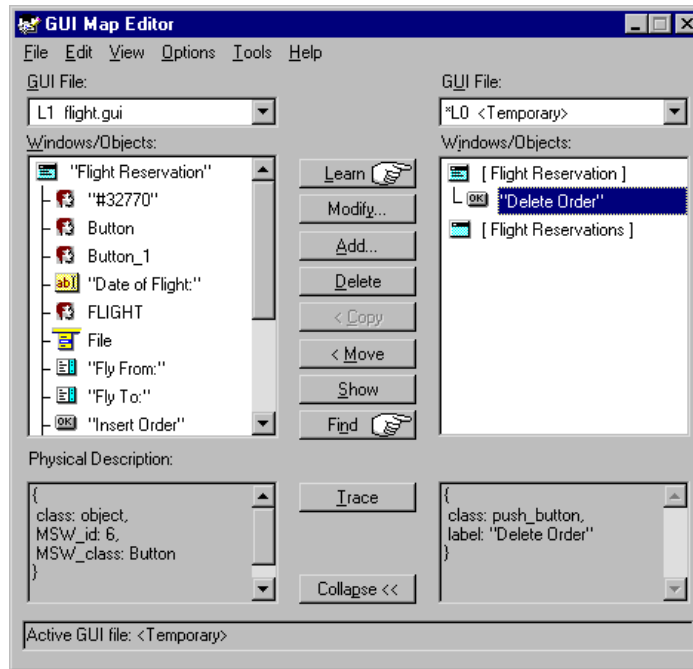
You can update GUI map files by copying or moving the description of GUI objects from one GUI map file to another. Note that you can only copy objects from a GUI file that you have opened for editing only, that is, from a file you have not loaded.

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.



- 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 context-sensitive Help.

Finding an Object in a GUI Map File

You can find a specific object in a GUI map file either by pointing to the object, or by selecting a line in your test script that contains the object.

To find an object from the application 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. The object is highlighted in the GUI map.

To find an object from the test script in a GUI map file:

- 1** Open an existing test and make sure that all relevant GUI maps are loaded.
- 2** Right-click anywhere in the line that contains the object and choose **Find In GUI Map**. The GUI Map Editor dialog box opens with the relevant object highlighted.

For more information on test scripts and the Test Script Language, refer to Chapter 7, “Enhancing Your Test Scripts with Programming” in the *Mercury WinRunner Advanced Features User’s Guide*.

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**.

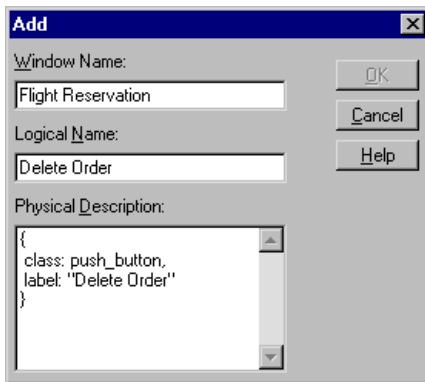
- 4** Display the contents of the file with the most recent description of the object by displaying the GUI map file in the GUI File box.
- 5** Select the object in the **Windows/Objects** box.
- 6** Click **Expand** to expand the GUI Map Editor dialog box.
- 7** Click **Trace**. The GUI map file in 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.



- 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**.

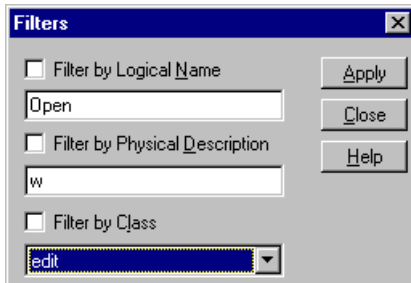
Filtering Displayed Objects

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.



- 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 context-sensitive Help.

Part III

Creating Tests—Basic

8

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
- ▶ Planning a Test
- ▶ 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 are displayed 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, or using the Function Viewer.

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, refer to Chapter 20, “Customizing the WinRunner User Interface” in the *Mercury WinRunner Advanced Features User’s Guide*.

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

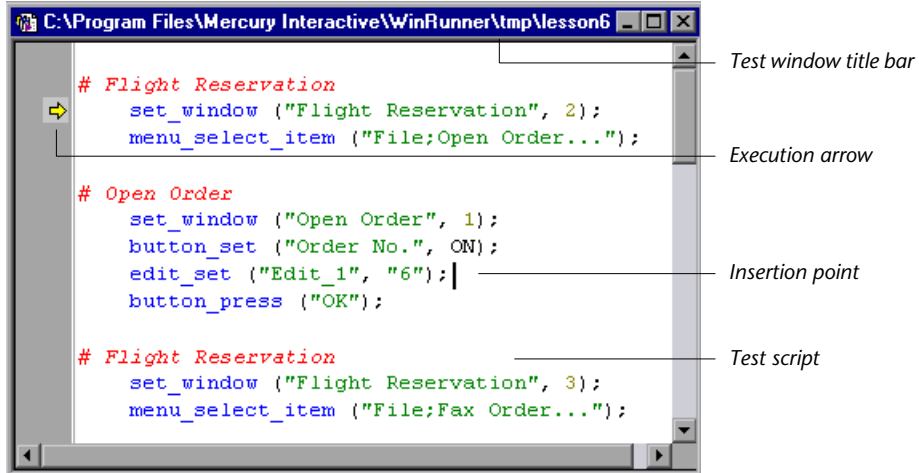
- 1** Decide on the functionality you want to test. Determine the checkpoints and synchronization points you need in the test script.
- 2** Document general information about the test in the Test Properties dialog box.
- 3** Choose a Record mode (*Context Sensitive* or *Analog*) and record the test on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



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 97 and “Creating Tests Using Analog Recording” on page 103.
- 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 107 and “Adding Synchronization Points to a Test” on page 108.
- 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 114.

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:

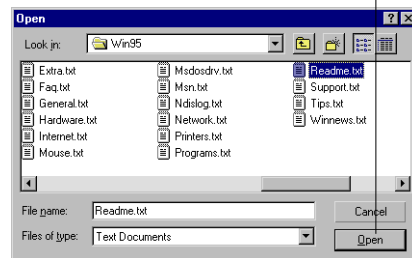
```
button_press ("Open");
```

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.



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).

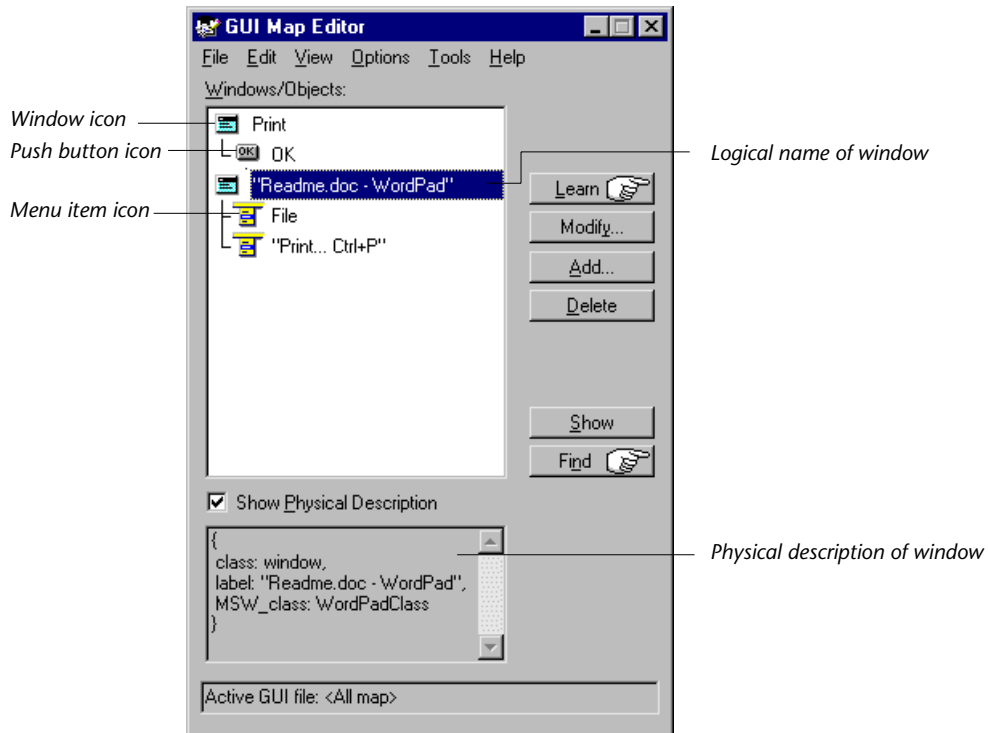
Remember that Context Sensitive tests work in conjunction with the GUI map and GUI map files. It is strongly recommended to read the “Introducing the GUI Map” section of this guide (beginning on page 23) before you start recording.

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:



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:



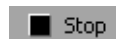
- 1** Choose **Test > Record–Context Sensitive** or click the **Record–Context Sensitive** button.



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 Chapter 2, “Configuring the GUI Map” in the <i>Mercury WinRunner Advanced Features User’s Guide</i> .
	You can add a custom GUI object class. For more information on creating custom GUI object classes and checking custom objects, refer to the <i>WinRunner Customization Guide</i> . 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 <i>WinRunner Customization Guide</i> .

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.
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 10, “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 menu 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, refer to Chapter 2, “Configuring the GUI Map” in the *Mercury WinRunner Advanced Features User’s Guide*. For information on using GUI checkpoints created in XRunner in WinRunner test scripts, see Chapter 9, “Checking GUI Objects.” For information on using bitmap checkpoints created in XRunner in WinRunner test scripts, see Chapter 15, “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, refer to Chapter 7, “Enhancing Your Test Scripts with Programming” in the *Mercury WinRunner Advanced Features User’s Guide*. For more information about the **invoke_application** function and an example of usage, refer to the *TSL Reference*. For more information on startup applications, refer to Chapter 22, “Setting Properties for a Single Test” in the *Mercury WinRunner Advanced Features User’s Guide*.

- 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, refer to Chapter 7, “Enhancing Your Test Scripts with Programming” in the *Mercury WinRunner Advanced Features User’s Guide*. 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, refer to Chapter 20, “Customizing the WinRunner User Interface” in the *Mercury WinRunner Advanced Features User’s Guide*.
- 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 23, “Setting Global Testing Options,” and Chapter 21, “Setting Testing Options from a Test Script” in the *Mercury WinRunner Advanced Features User’s Guide*.
- ▶ WinRunner supports recording and running tests on applications with drop-down and menu-like toolbars. 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 9, “Checking GUI Objects,” for more information.
- ▶ Bitmap checkpoints take a “snapshot” of a window or area of your application and compare this to an image captured in an earlier version. See Chapter 15, “Checking Bitmaps,” for more information.
- ▶ Text checkpoints read text in GUI objects and in bitmaps and enable you to verify their contents. See Chapter 16, “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 14, “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 18, “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 19, “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.

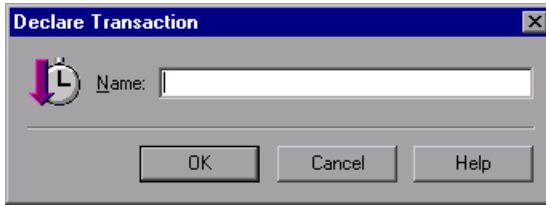
You can insert **declare_transaction**, **start_transaction**, and **end_transaction** statements manually, or you can use the **Insert > Transactions** options to insert these statements.

To insert transaction statements using the Insert > Transactions options:

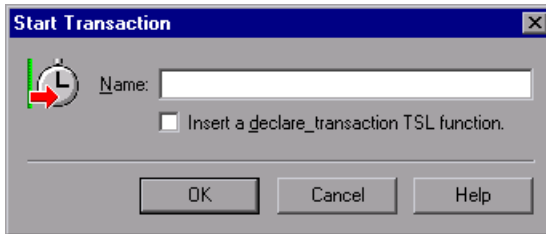
- 1** If you want to insert the **declare_transaction** and **start_transaction** statements on consecutive lines, proceed to step 4.

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.

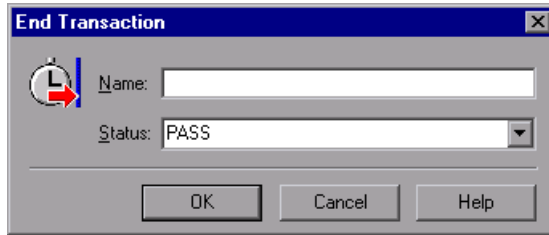


- 3 Enter a name for the transaction and click OK. The **declare_transaction** statement is added to your test.
- 4 Place the cursor at the beginning of the line where you want the transaction measurement to begin.
- 5 Choose **Insert > Transactions > Start Transaction**. The Start Transaction dialog 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, refer to Chapter 20, "Customizing the WinRunner User Interface" in the *Mercury WinRunner Advanced Features User's Guide*.

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.
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 SCREEN AREA	Alt Right + F11	Captures text in a specified area and adds a get_text statement to the test script.
INSERT FUNCTION FOR OBJECT/WINDOW	F8	Inserts a TSL function for a GUI object.
INSERT FUNCTION FROM FUNCTION GENERATOR	F7	Opens the Function Generator dialog box.
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, refer to Chapter 8, “Generating Functions” in the *Mercury WinRunner Advanced Features User’s Guide*.

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 the *Mercury WinRunner Advanced Features User’s 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.

Edit Command	Description
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.
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, refer to Chapter 19, "Customizing the Test Script Editor" in the <i>Mercury WinRunner Advanced Features User's Guide</i> .
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, refer to Chapter 19, "Customizing the Test Script Editor" in the <i>Mercury WinRunner Advanced Features User's Guide</i> .
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.

Note: To create a new scripted component, you follow the above instructions to create a test and then save the document as a scripted component. For more information, refer to the *Mercury WinRunner Advanced Features User's Guide*.

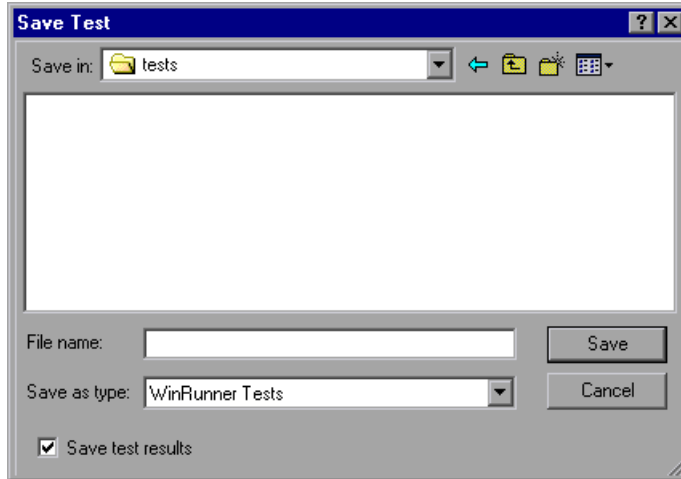
Saving a Test

The following options are available for saving tests:

- ▶ Save changes to a previously saved test by choosing **File > Save** or by clicking **Save** on the toolbar.
- ▶ Save a new test to the file system or to Quality Center by choosing **File > Save As Test** or by clicking **Save** on the toolbar.
- ▶ Save two or more open tests simultaneously by choosing **File > Save All**.
- ▶ Save a test script as a scripted component in a Quality Center project by choosing **File > Save As Scripted Component**. For more information, refer to the *Mercury WinRunner Advanced Features User's Guide*.

To save a new test to the file system:

- 1 On the **File** menu, choose the **Save** or **Save as Test** command, or click **Save** on the toolbar. The Save Test dialog box opens.



- 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.

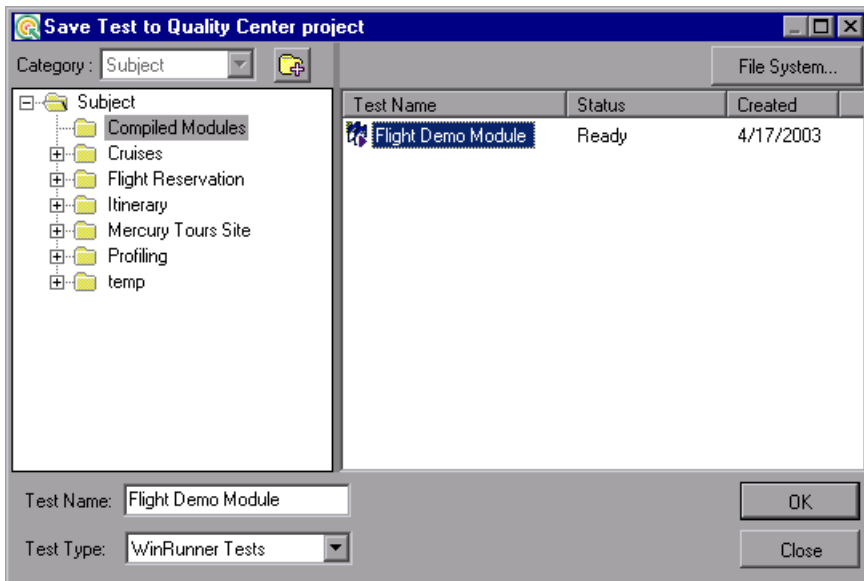
Note: By default, this option is selected when saving a new test (**Save**), and cleared when saving an existing test under a new name (**Save As**).

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

To save a test to a Quality Center project:

Note: You can save a test to a Quality Center database only if you are connected to a Quality Center project. For additional information, refer to Chapter 26, “Managing the Testing Process” in the *Mercury WinRunner Advanced Features User’s Guide*.

- 1 After connecting to a Quality Center project, choose **File > Save as Test**. The Save Test to Quality Center Project dialog box opens.



The test plan tree from the Quality Center Test Plan module is displayed.

Note that the **Save Test to Quality Center Project** dialog box opens only when WinRunner is connected to a Quality Center project.



- 2 Select the relevant subject folder in the test plan tree or click the **New Folder** button to create a new 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 Quality Center, or refresh the test plan tree in the Test Plan module, the new test will be displayed in the tree. Refer to the *Mercury Quality Center User's Guide* for more information.

For more information on saving tests to a Quality Center project, refer to Chapter 26, "Managing the Testing Process" in the *Mercury WinRunner Advanced Features User's Guide*.

Opening an Existing Test

You can open an existing test from the file system or from a Quality Center project.

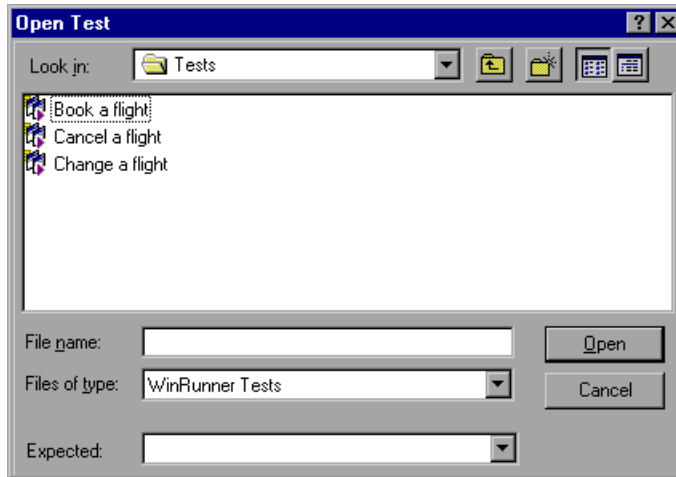
You can also open a scripted component from a Quality Center project. For more information, see "Opening an Existing Scripted Component" on page 122.

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

To open a test from the file system:

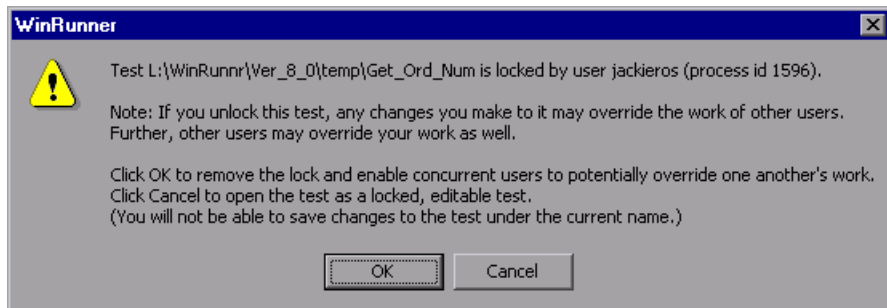


- 1 Choose **File > Open** or click **Open** to open the Open Test dialog box.



- 2 In the **Look in** box, click the location of the test you want to open.
- 3 In the **File name** box, click the name of the test to open.
- 4 If the test has more than one set of expected results, click the folder you want to use on the **Expected** list. The default folder is called *exp*.
- 5 Click **Open** to open the test.

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



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.

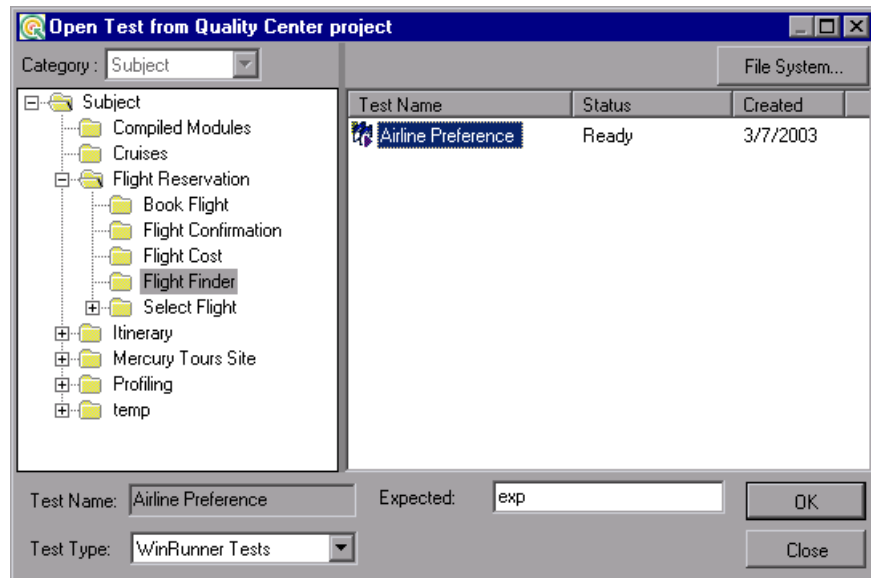
Click **OK** to unlock the test only if you are sure that your work will not interfere with other users.

To open a test from a Quality Center project:

Note: You can open a test from a Quality Center database only if you are connected to a Quality Center project. For additional information, refer to Chapter 26, “Managing the Testing Process” in the *Mercury WinRunner Advanced Features User’s Guide*.



- 1 Choose **File > Open Test** or click **Open**. If you are connected to a Quality Center project, the Open Test from Quality Center Project dialog box opens and displays the test plan tree.



Note that the **Open Test from Quality Center Project** dialog box opens only when WinRunner is connected to a Quality Center project.

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

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

- 3 Select a test in the **Test Name** list. The test is displayed 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 Quality Center project, refer to Chapter 26, “Managing the Testing Process” in the *Mercury WinRunner Advanced Features User's Guide*.

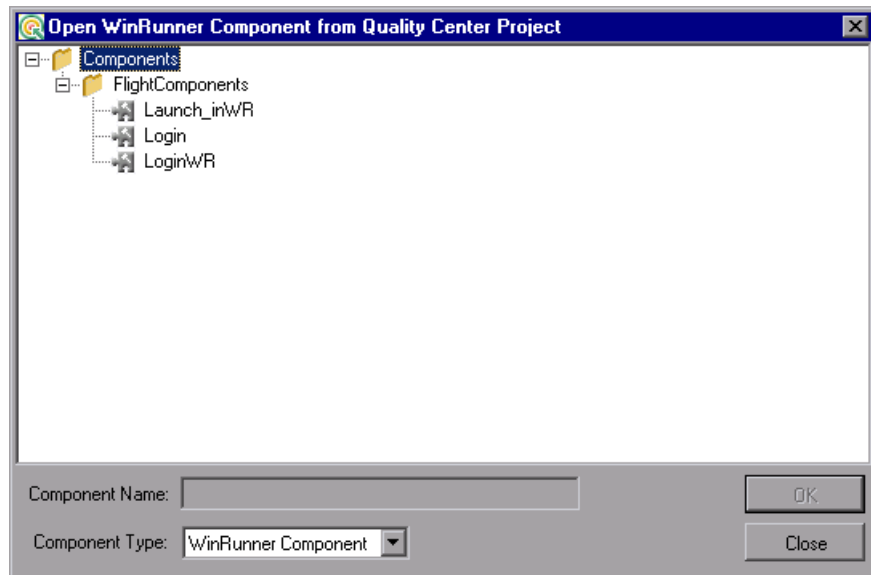
Opening an Existing Scripted Component

WinRunner Scripted components can be included in business process tests in Quality Center with Business Process Testing support. However, they cannot be edited in Quality Center. You can open an existing WinRunner scripted component in WinRunner for viewing or editing if required.

To open a scripted component from a Quality Center project:

Note: You can open a scripted component from a Quality Center database only if you are connected to a Quality Center project. For additional information, refer to Chapter 26, “Managing the Testing Process” in the *Mercury WinRunner Advanced Features User’s Guide*.

- 1 After connecting to a Quality Center project, choose **File > Open Scripted Component** or press CTRL+H. The Open WinRunner Component from Quality Center Project dialog box opens and displays the component tree.



Note: The Open Scripted Component option in the File menu is visible only when you are connected to Quality Center with Business Process Testing support.

- 2 Select the relevant component in the component tree. To expand the tree and view sublevels, double-click closed folders. To collapse the tree, double-click open folders. The scripted component is displayed in the read-only **Component Name** box.
- 3 Click **OK** to open the scripted component. The component opens in a window in WinRunner. Note that WinRunner's title bar shows the full subject path of the scripted component.
- 4 View or edit the component as required.

For more information on opening scripted components in a Quality Center project, refer to Chapter 26, “Managing the Testing Process” in the *Mercury WinRunner Advanced Features User's Guide*.

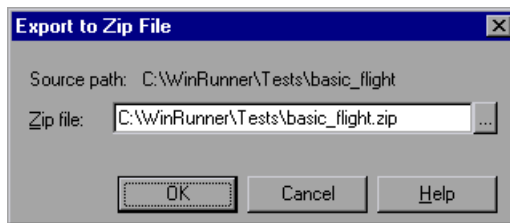
Zipping and Extracting WinRunner Tests

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:

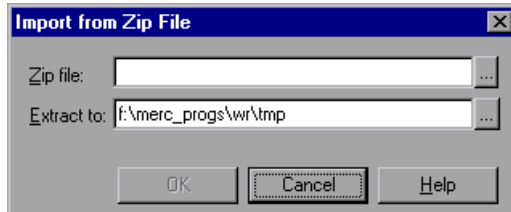
- 1 Open the test you want to zip.
- 2 If the open test contains unsaved changes, save the test.
- 3 Choose **File > Export to Zip File**. The Export to Zip File dialog box opens and displays the source path of the test and a suggested zip file name.



- 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.



- 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**.

9

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.

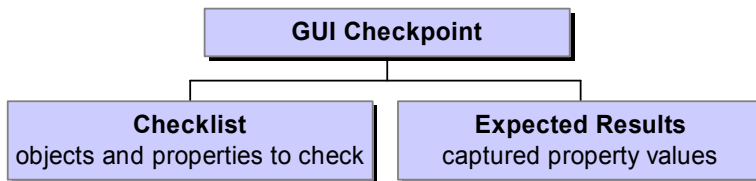
This chapter describes:

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

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 21, “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, refer to Chapter 6, “Using Regular Expressions” in the *Mercury WinRunner Advanced Features User’s Guide*.

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 11, “Working with ActiveX and Visual Basic Controls.” For information on WinRunner support for PowerBuilder, see Chapter 12, “Checking PowerBuilder Applications.”

You can also create GUI checkpoints that check the contents and properties of tables. For information, see Chapter 13, “Checking Table Contents.”

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 579.

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 562.

Checking a Single Property Value

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

button_check_info	scroll_check_info
edit_check_info	static_check_info
list_check_info	win_check_info
obj_check_info	

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

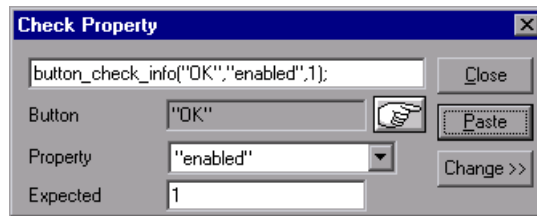
To create a GUI checkpoint for a property value:

- 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.



- 3 You can modify the arguments for the property check.
 - To modify assigned argument values, choose a value from the **Property** 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, refer to Chapter 8, “Generating Functions” in the *Mercury WinRunner Advanced Features User’s Guide*.

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 158.

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 140.

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.

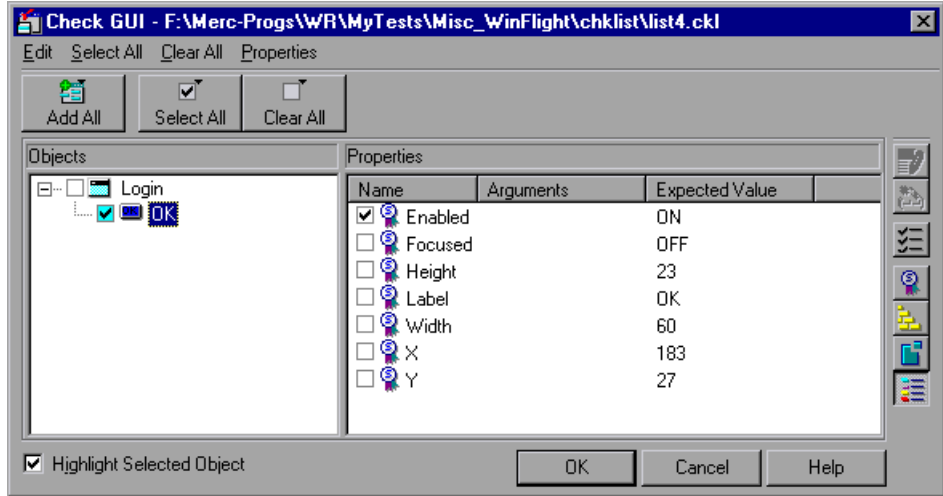
To create a GUI checkpoint by specifying which properties to check:



- 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.



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 170.



- ▶ 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) is displayed 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 specify arguments only 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 164.

- ▶ 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 150.

- 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 140.

For more information on the Check GUI dialog box, see “Understanding the GUI Checkpoint Dialog Boxes” on page 148.

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 158.

To create a GUI checkpoint for two or more objects:



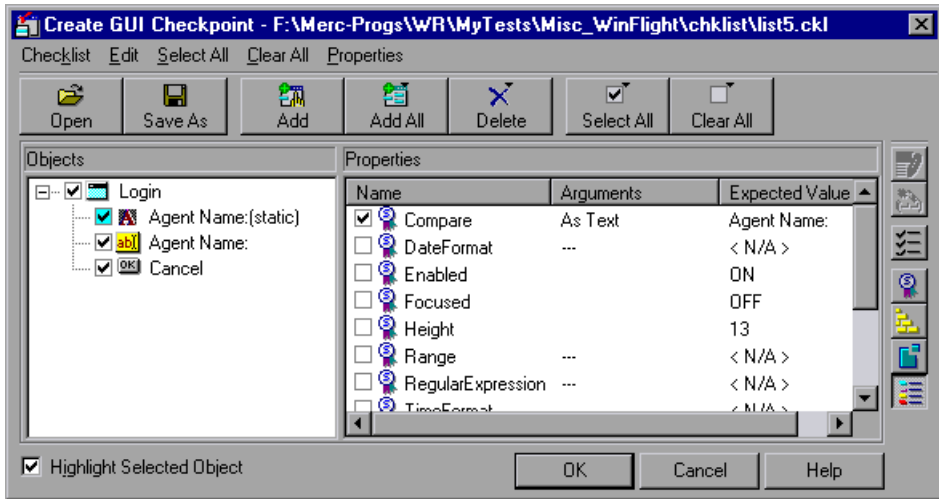
- 1 Choose **Insert > GUI Checkpoint > For Multiple Objects** or click the **GUI Checkpoint for Multiple Objects** button on the User toolbar. If you are recording in Analog mode, press the CHECK GUI FOR MULTIPLE OBJECTS softkey in order to avoid extraneous mouse movements. The Create GUI Checkpoint 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 137.
- 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.

- Click the right mouse button to stop the selection process and to restore the mouse pointer to its original shape. The Create GUI Checkpoint dialog box reopens.



- The Objects 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 170.



- 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 is displayed 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 specify arguments only 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 164.

- ▶ 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 152.
- 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 140.

For more information on the Create GUI Checkpoint dialog box, see “Understanding the GUI Checkpoint Dialog Boxes” on page 148.

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 158.

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:

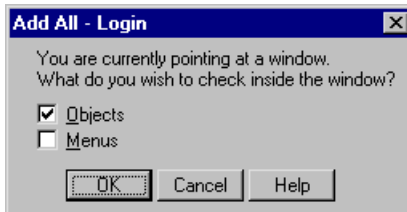


- 1 Choose **Insert > GUI Checkpoint > For Object/Window**, or click the **GUI Checkpoint for Object/Window** button on the User toolbar. If you are recording in Analog mode, press the CHECK GUI FOR OBJECT/WINDOW softkey in order to avoid extraneous mouse movements. Note that you can press the CHECK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



- 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 150.

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 141.
- ▶ 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 438. 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, refer to Chapter 21, “Setting Testing Options from a Test Script” in the *Mercury WinRunner Advanced Features User’s Guide*.

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, refer to Chapter 21, “Setting Testing Options from a Test Script” in the *Mercury WinRunner Advanced Features User’s Guide*.

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.

To enable 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 59.

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 143.

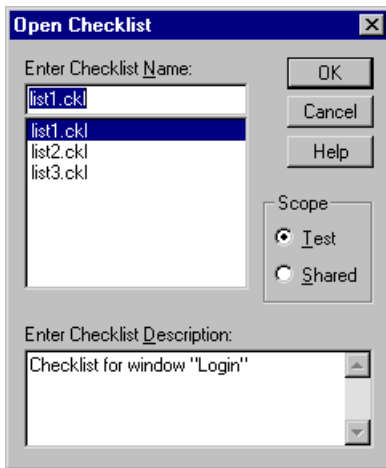
To use an existing GUI checklist in a GUI checkpoint:



- 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**.



- 4 Select a checklist and click **OK**.

The Open Checklist dialog box closes and the selected list is displayed 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 172.

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.

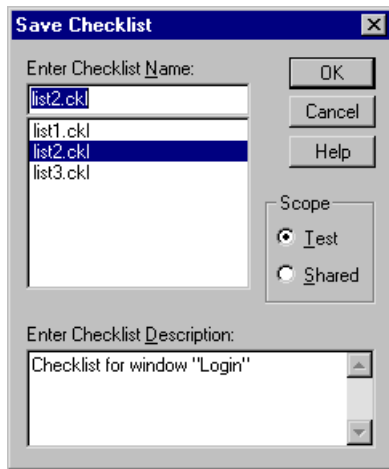
The default folder in which WinRunner stores your shared checklists is *WinRunner installation folder/chklist*. To choose a different folder, you can use the **Shared checklists** box in the **Folders** category of the General Options dialog box. For more information, see Chapter 23, “Setting Global 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 297.

- 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.



- 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:

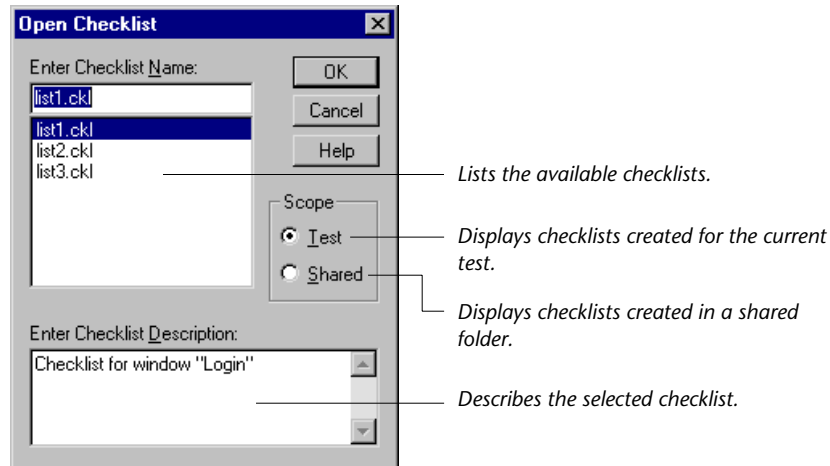
- change which objects in a window to check
- change which properties of an object to check
- change the arguments for an existing property check
- specify the arguments for a new property check

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 59.

To edit an existing GUI checklist:

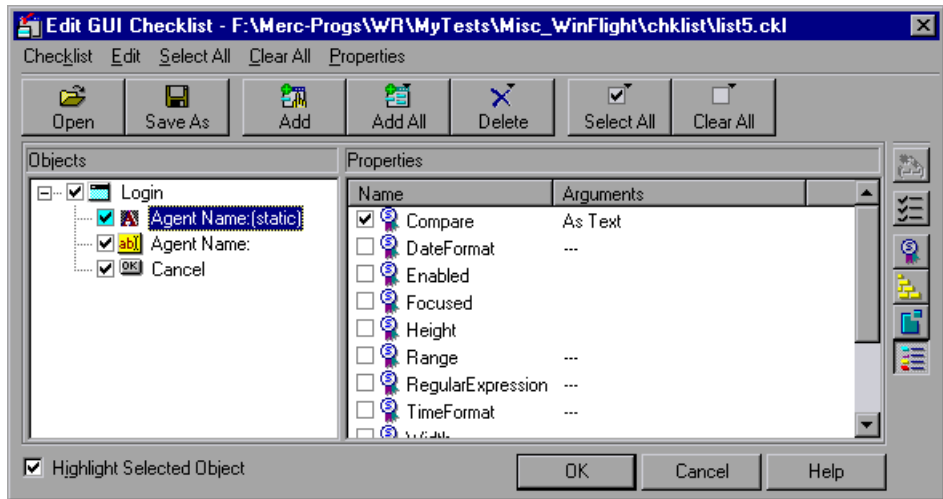
- 1** Choose **Insert > Edit GUI Checklist**. The Open Checklist dialog box opens.
- 2** A list of checklists for the current test is displayed. If you want to see checklists in a shared folder, click **Shared**.

For more information on sharing GUI checklists, see “Saving a GUI Checklist in a Shared Folder” on page 143.



- 3** Select a GUI checklist.

- 4 Click **OK**. The Open Checklist dialog box closes. The Edit GUI Checklist dialog box opens and displays the selected checklist.



- 5 To see a list of the properties to check for a specific object, click the object name in the **Objects** pane. The **Properties** pane lists all the properties for the selected object. To change the viewing options for the properties for an object, use the Show Properties buttons. For more information, see “The Edit GUI Checklist Dialog Box,” on page 155.

- ▶ 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 is displayed 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 specify arguments only 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 164.

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 148.

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 429.

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 be displayed in the GUI Checkpoint dialog boxes:

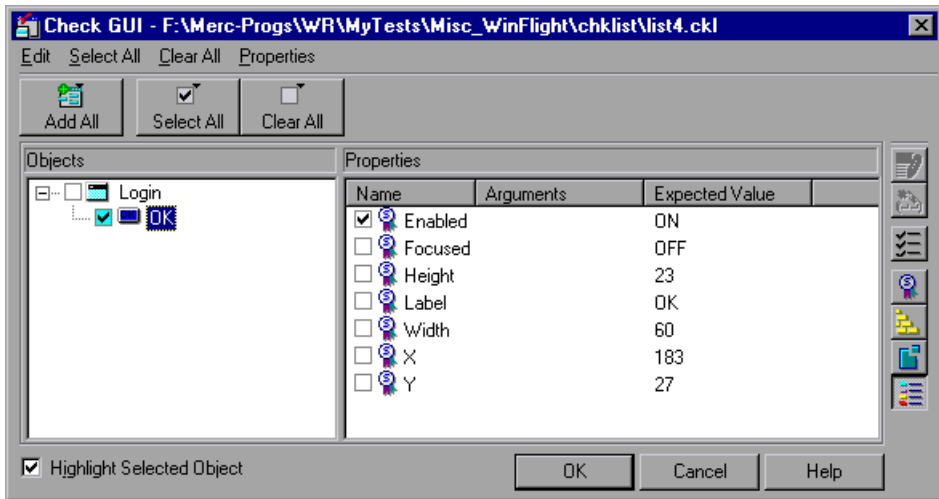
Message	Meaning	Dialog Box	Location
Complex Value	The expected or actual value of the selected property check is too complex to display in the column. This message often appears for content checks on tables.	Check GUI, Create GUI Checkpoint, GUI Checkpoint Results* (see note below)	Properties pane, Expected 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 172 or Chapter 21, “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.














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 <i>ui_function</i> parameter that is defined only for classes customized using the <code>gui_ver_add_class</code> function. Note that this button appears only if at least one object in the Objects pane belongs to a class for which the <i>ui_function</i> parameter has been defined using the <code>gui_ver_add_class</code> function. For additional information, refer to the <i>WinRunner Customization Guide</i> .
 Edit Expected Value	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 170.
 Specify Arguments	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 164.
 Show Selected Properties Only	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.

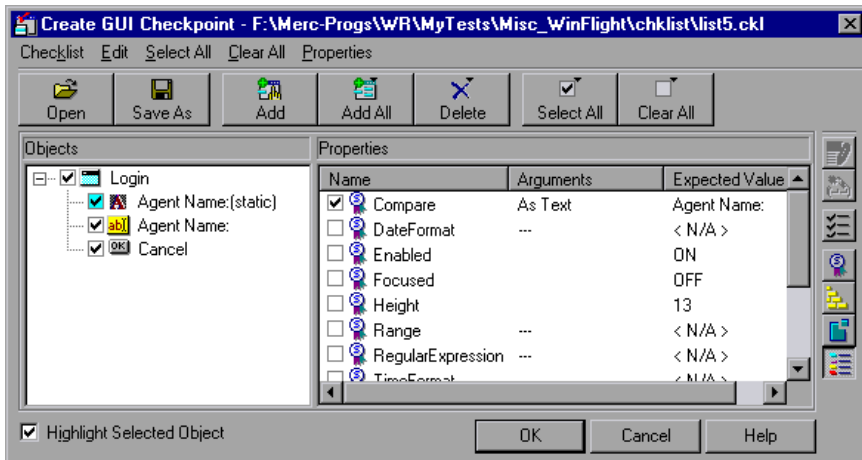
Button	Description
	Show Standard Properties Only displays only standard properties.
	Show Nonstandard Properties Only displays only nonstandard properties, such as Visual Basic, PowerBuilder, and ActiveX control properties.
	Show User Properties Only displays only user-defined property checks. To create user-defined property checks, refer to the <i>WinRunner Customization Guide</i> .
	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.


















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

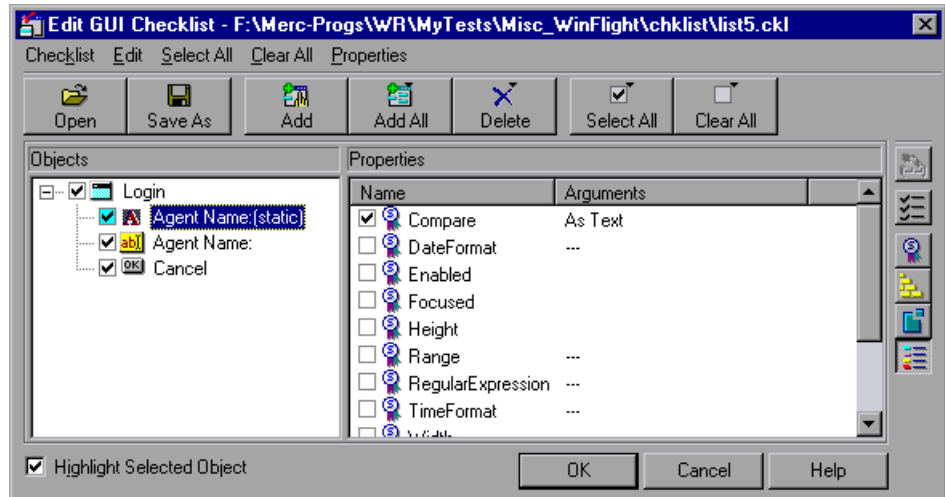
Button	Description
	<p>Clear All clears all objects, properties, or objects of a given class in the Create GUI Checkpoint dialog box. If you want to clear all objects of a given class, the Classes of Objects dialog box opens. Specify the class of objects to clear.</p>
	<p>Property List calls the <i>ui_function</i> 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 <i>ui_function</i> parameter has been defined using the gui_ver_add_class function. For additional information, refer to the <i>WinRunner Customization Guide</i>.</p>
	<p>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 170.</p>
	<p>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 164.</p>
	<p>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.</p>
	<p>Show Standard Properties Only displays only standard properties.</p>
	<p>Show Nonstandard Properties Only displays only nonstandard properties, such as Visual Basic, PowerBuilder, and ActiveX control properties.</p>
	<p>Show User Properties Only displays only user-defined property checks. To create user-defined property checks, refer to the <i>WinRunner Customization Guide</i>.</p>
	<p>Show All Properties displays all properties, including standard, nonstandard, and user-defined properties.</p>

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**.








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 <i>ui_function</i> 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 <i>ui_function</i> parameter has been defined using the gui_ver_add_class function. For additional information, refer to the <i>WinRunner Customization Guide</i> .
	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 164.

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, 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 <i>WinRunner Customization Guide</i> .
	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 429.

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 164.

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 166.

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, refer to Chapter 6, “Using Regular Expressions” in the *Mercury WinRunner Advanced Features User’s Guide*.

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.

ItemCount: 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.

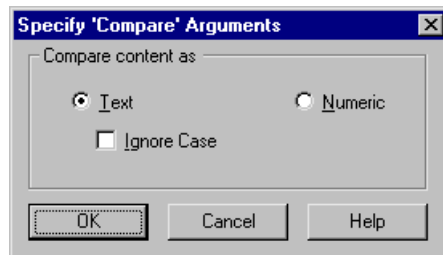
A dialog box for the selected property check opens.

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.

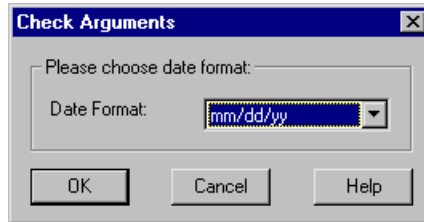


- Click **Text** to check the contents as text (default setting).
- To ignore the case when checking text, select the **Ignore Case** check box.
- Click **Numeric** to check the contents as a number.

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

DateFormat Property Check

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



Date Formats

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

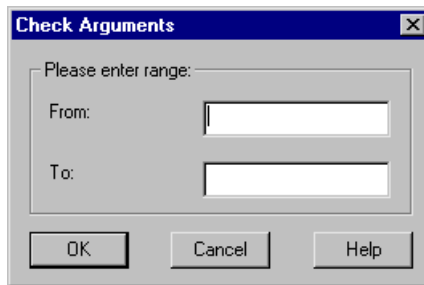
mm/dd/yy	09/24/04
dd/mm/yy	24/09/04
dd/mm/yyyy	24/09/2004
yy/dd/mm	04/24/09
dd.mm.yy	24.09.04
dd.mm/yyyy	24.09.2004
dd-mm-yy	24-09-04
dd-mm-yyyy	24-09-2004
yyyy-mm-dd	2004-09-24
Day, Month dd, yyyy	Friday (or Fri), September (or Sept) 24, 2004
dd Month yyyy	24 September 2004
Day dd Month yyyy	Friday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

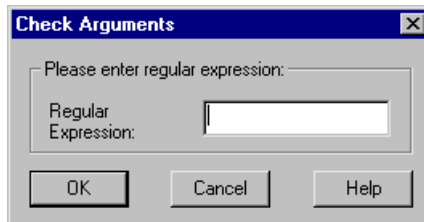
Checks that the contents of the edit class or static_text class object are within the specified range. In the Check Arguments dialog box, specify the lower limit in the top edit field, and the upper limit in the bottom edit field.

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



RegularExpression Property Check

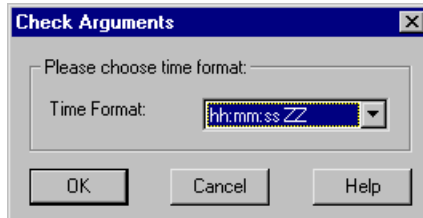
Checks that the string in the edit class or static_text class object meets the requirements of the regular expression. In the Check Arguments dialog box, enter a string into the Regular Expression box. You do not need to precede the regular expression with an exclamation point. For more information, refer to Chapter 6, “Using Regular Expressions” in the *Mercury WinRunner Advanced Features User’s Guide*.



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.

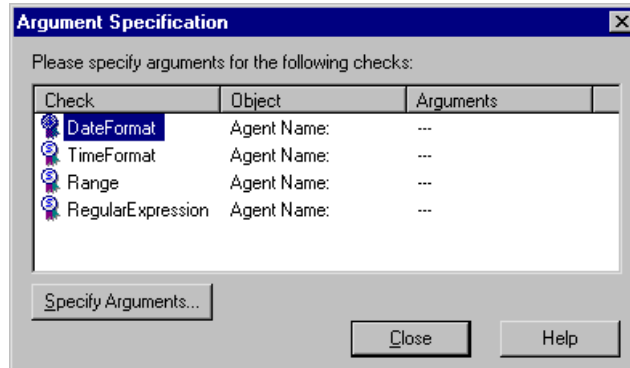
Specifying Arguments for One Property Check

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:



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 172.

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.

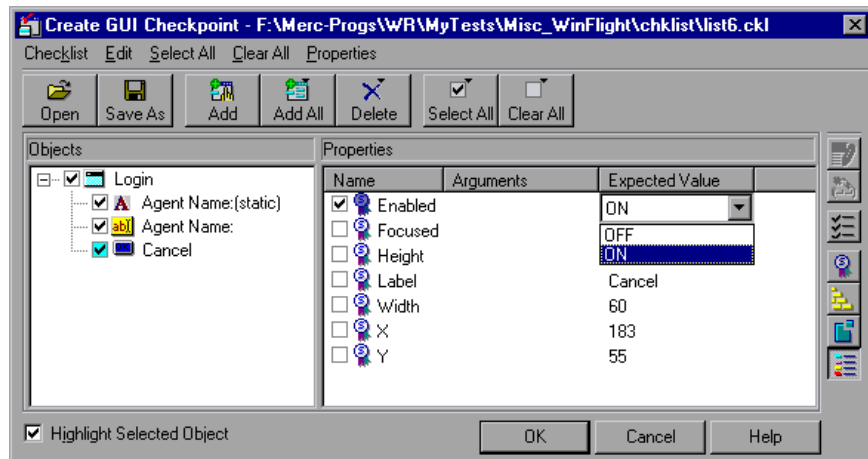
- 2 If the Check GUI dialog box or the Create GUI Checkpoint dialog box is not already open, choose **Insert > GUI Checkpoint > For Multiple Objects** to open the **Create GUI Checkpoint** dialog box and click **Open** to open the checklist in which to edit the expected value. Note that the Check GUI 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:



- 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:



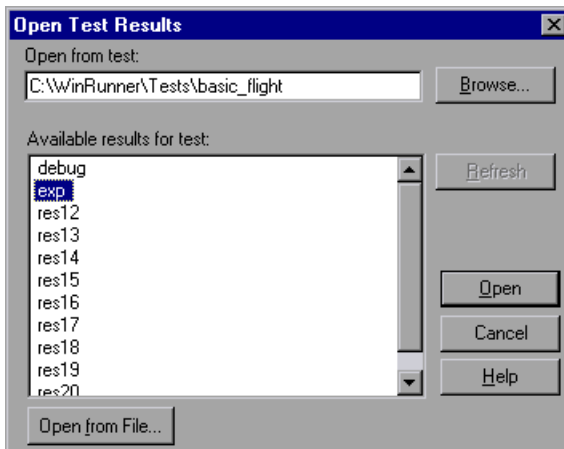
- 1 Choose **Tools > Test Results** or click **Test Results**.

The WinRunner Test Results window opens.

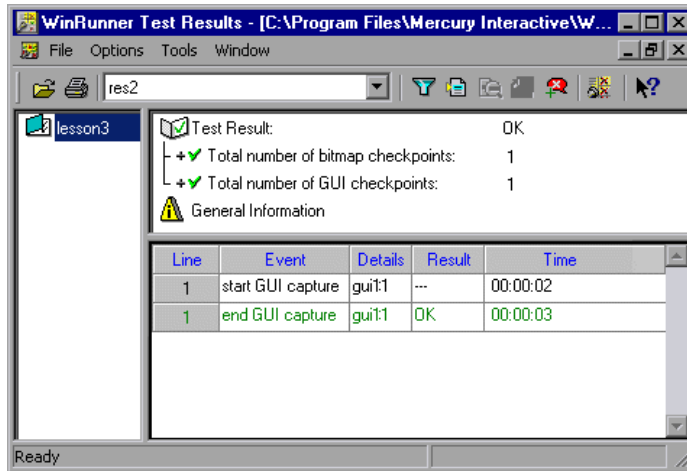
- 2 Display the expected results:



- 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**.



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

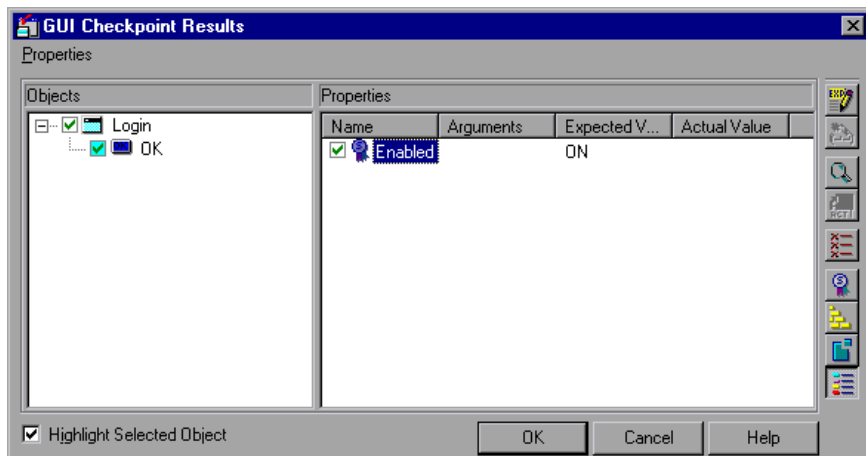


- 3 Locate the GUI checkpoint by looking for **end GUI capture** events.



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 170.

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 499.

For more information on working in the Test Results window, see Chapter 21, “Analyzing Test Results.”

10

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
- ▶ Viewing Recorded Web Object Properties
- ▶ Using Web Object Properties in Your Tests
- ▶ Checking Web Objects

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 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” on page 178.

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, refer to Chapter 2, “Configuring the GUI Map” in the *Mercury WinRunner Advanced Features User’s Guide*. 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 187.

For more information on descriptive programming, refer to Chapter 7, “Enhancing Your Test Scripts with Programming” in the *Mercury WinRunner Advanced Features User’s Guide*.

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 180.
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.
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.

Property Name	Description
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, refer to Chapter 7, “Enhancing Your Test Scripts with Programming” in the *Mercury WinRunner Advanced Features User’s Guide*.

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 *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.
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 containing 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 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
current_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 <code>name</code> property.
rows	The number of rows in the table.
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.

Property Name	Description
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 <code><input></code> 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
kind	The type of edit box. Possible values: single-line multi-line

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 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: <code><input type=text></code>

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 **html_combobox** MSW classes:

Property Name	Description
element_name	The name property specified within the <code><SELECT></code> 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 9, "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.

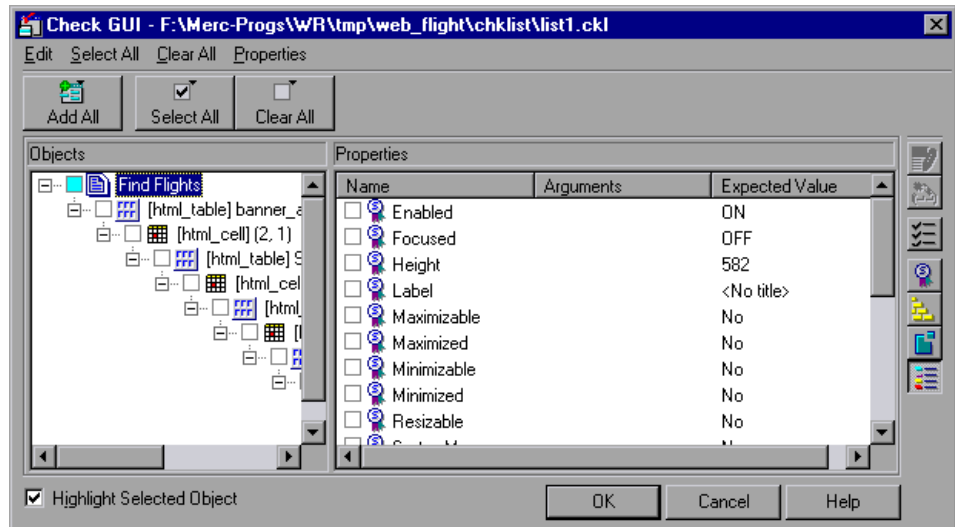
To check standard frame properties:



- 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.



- 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**, **Maximizable**, **Minimized**, **Maximized** these properties are not relevant for frame objects.
- **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*.

Checking the Object Count in Frames

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

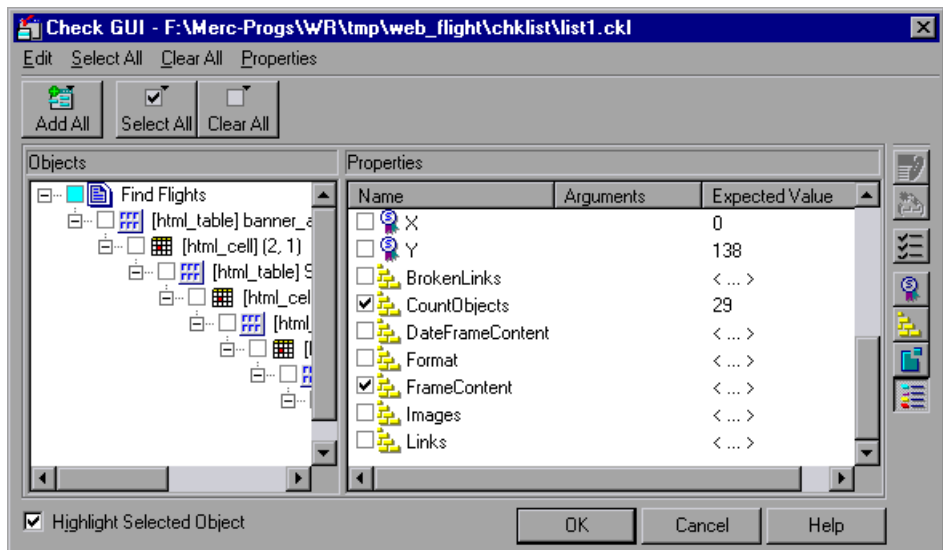
To check the object count in a frame:



- 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.



- 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*.

Checking the Structure of Frames, Tables, and Cells

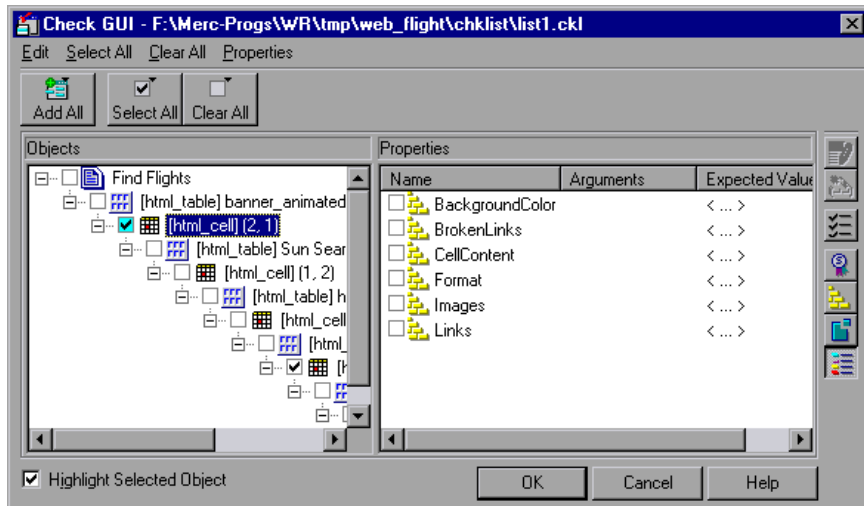
You can create a GUI checkpoint to check the structure of frames, tables, and cells on a Web page.

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.



- 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*.

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 203.

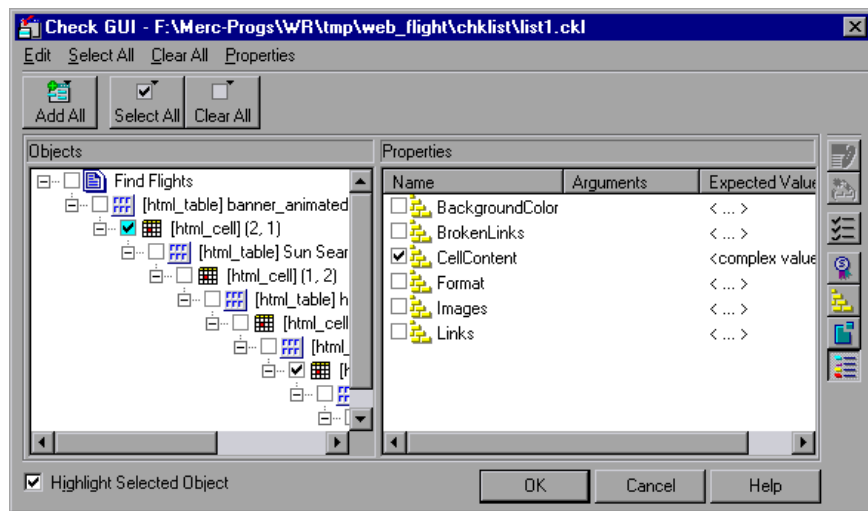
To check content:



- 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.



- 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*.

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.

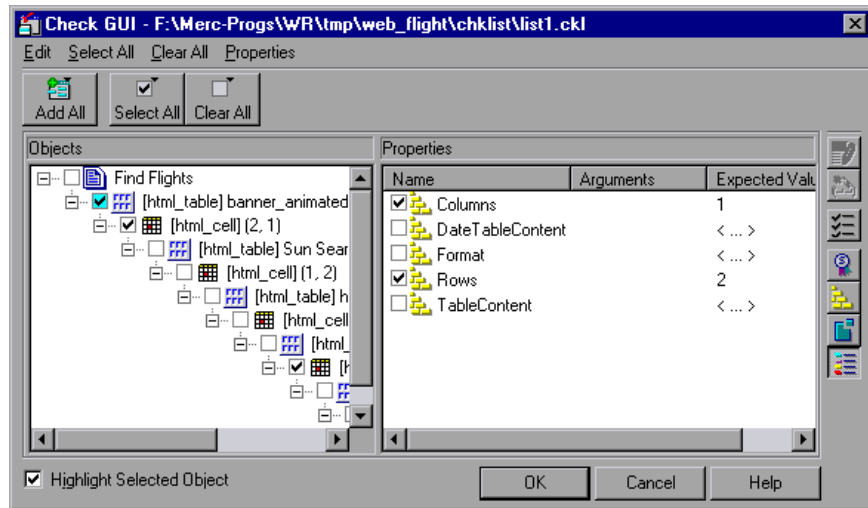
To check the number of columns and rows in a table:



- 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.



- 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*.

Checking the URL of Links

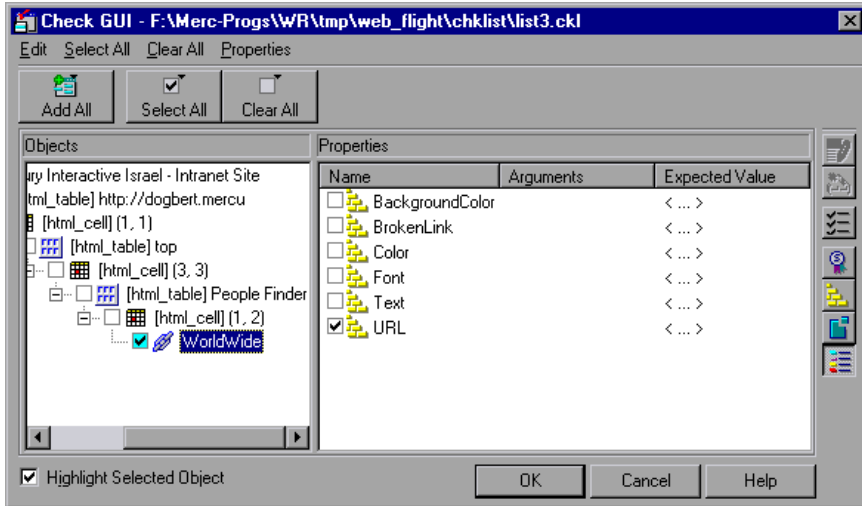
You can create a GUI checkpoint to check the URL of a text link or an image link in your Web page.

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.



- 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*.

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.

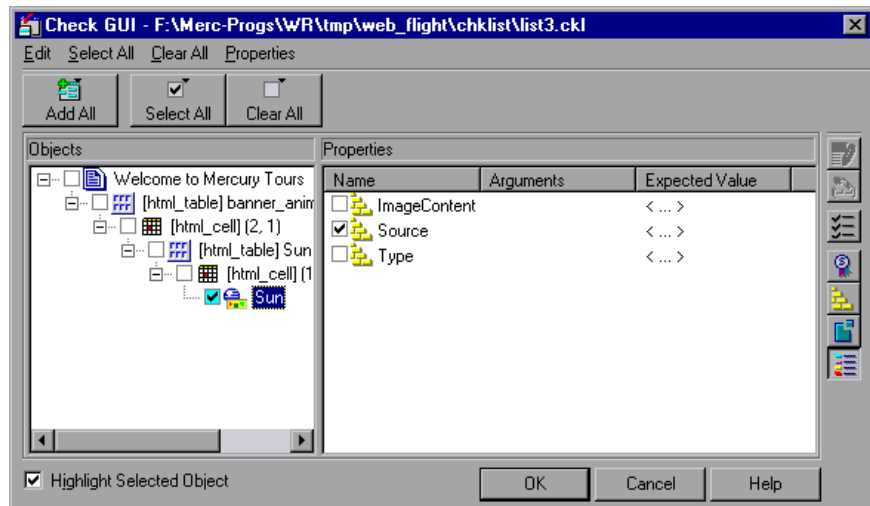
To check the source or type of an image or an image link:



- 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.



- 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*.

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.

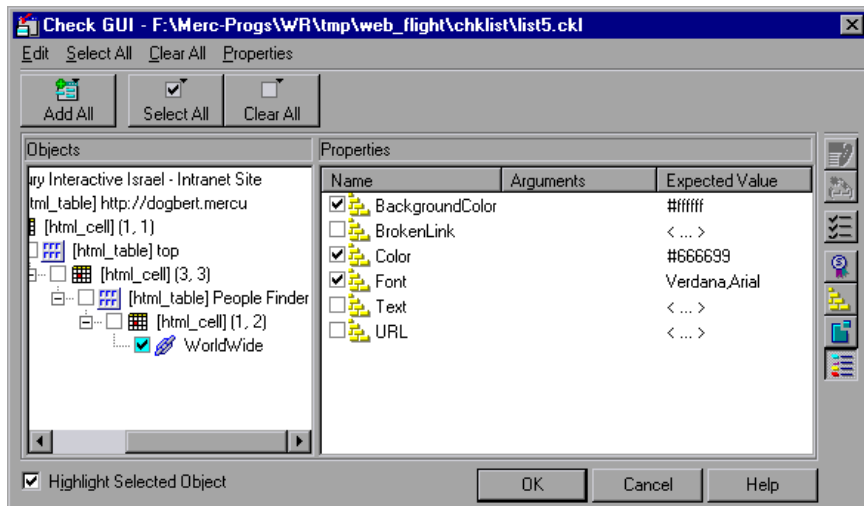
To check the color or font of a text link:



- 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.



- 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*.

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.

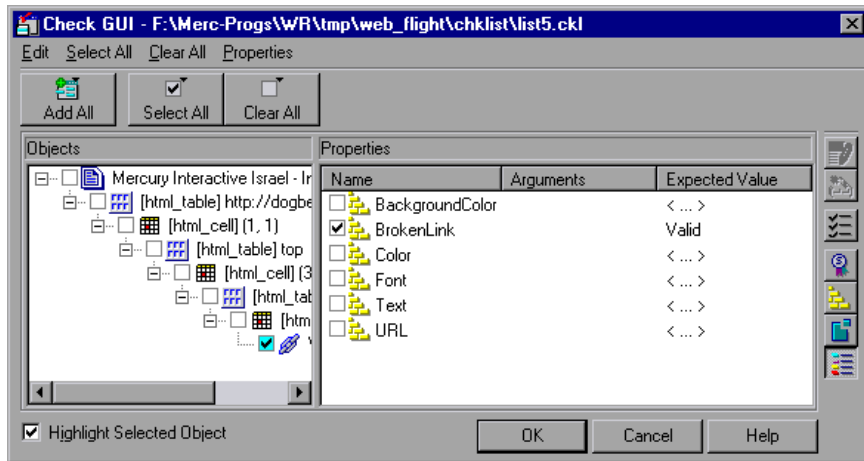
To check a single broken link:



- 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.



- 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.

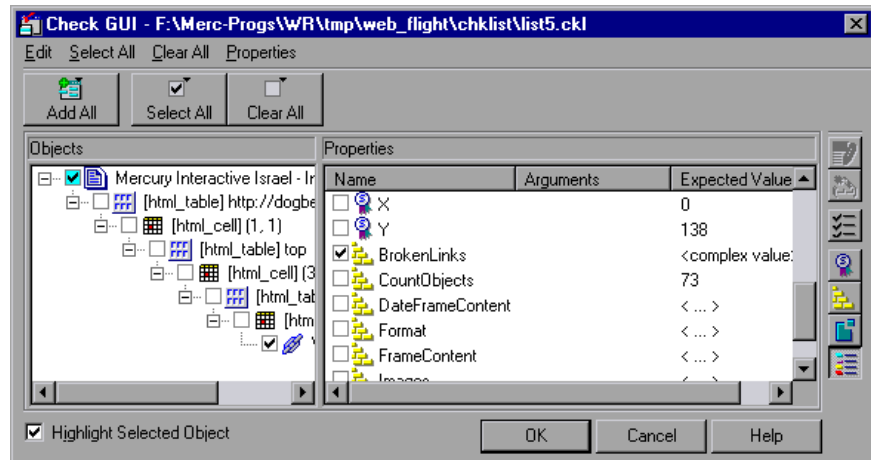
WinRunner captures the object information and stores it in the test's expected results folder. The WinRunner window is restored and a checkpoint appears in your test script as an **obj_check_gui** or **win_check_gui** statement. For more information on the **obj_check_gui** and **win_check_gui** function,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

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.



- 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 205.

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*.

Checking Links and Images in a Frame

You can create a checkpoint to check image links, text links and images in a frame.

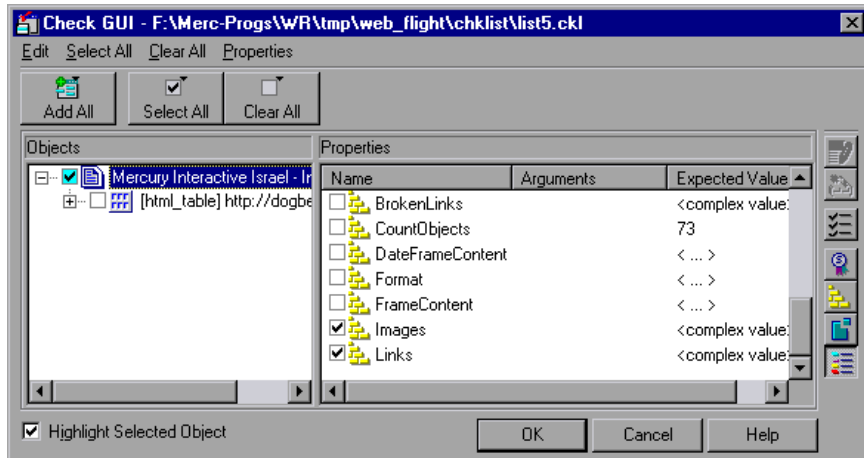
To check links and images in a frame:



- 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.



- 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 205.

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*.

Checking the Text Content of Tables

You can create a checkpoint to check the text content of a table.

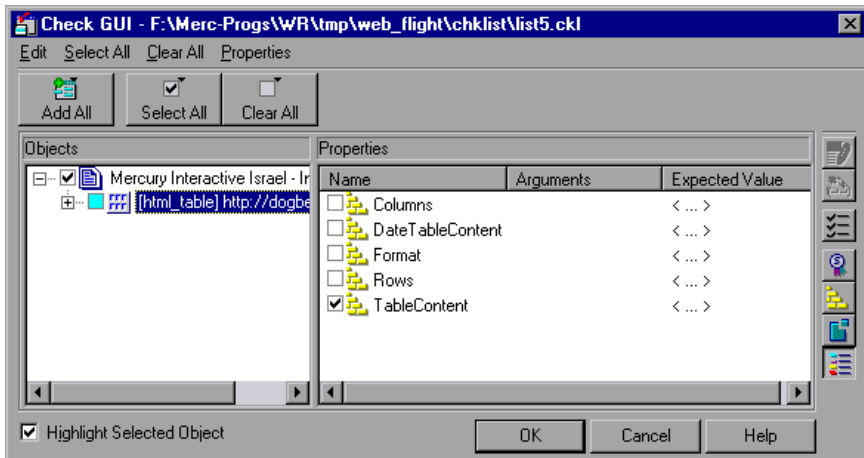
To check the content of a table:



- 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.



- 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 205.

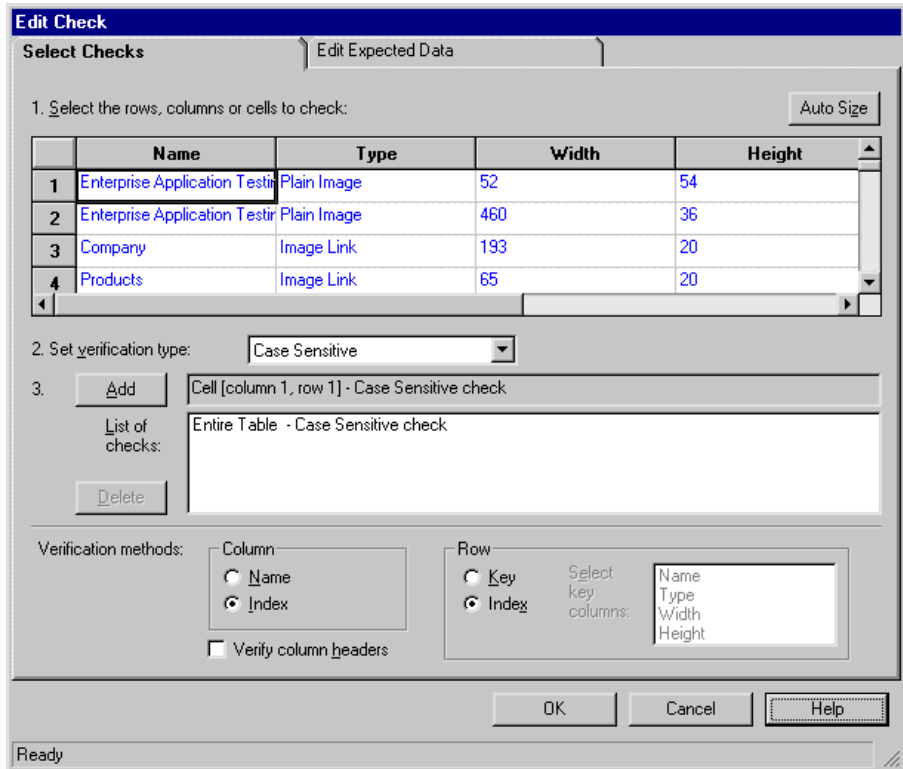
When you are done, click **OK** to save and close the Edit Check dialog box. The Check GUI dialog box is restored.

- 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*.

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.



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 209.

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 210.

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 206. Choosing this option enables the **Verify column headers** check box, which enables you to check column headers as well as cells.

► **Row:**

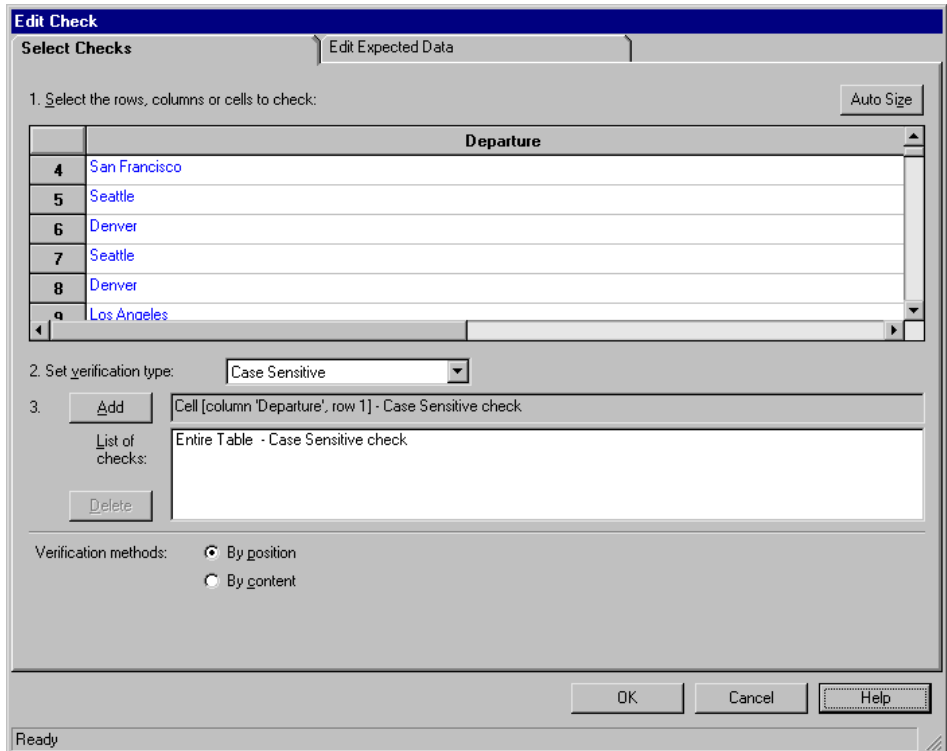
- **Key:** WinRunner looks for the rows in the selection according to the data in the key column(s) specified in the **Select key columns** list box. For example, you could tell WinRunner to identify the second row in the table on page 211 based on the arrival time for that row. A shift in the position 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.



- **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	Airline	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*. For more information on checking text in a non-Web object, see Chapter 16, “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.

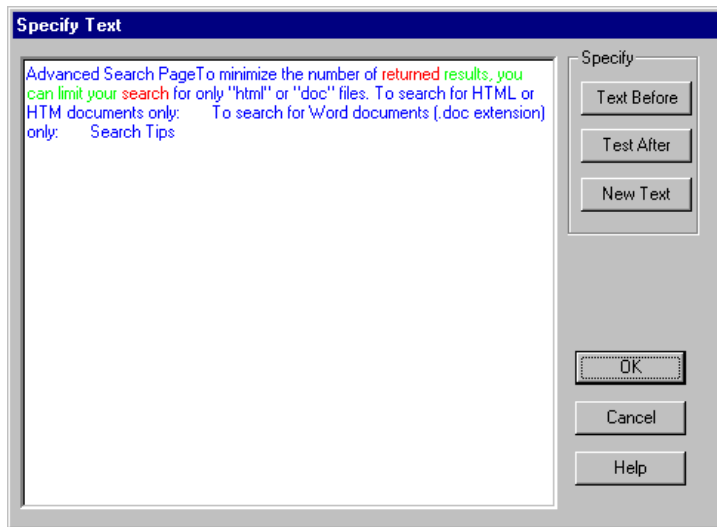
To read a text string from a frame or an object:

- 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.



The text string to be read is displayed in green and underlined. The bold 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 underlined and in green. The text that appears before and after your text string is displayed bold 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

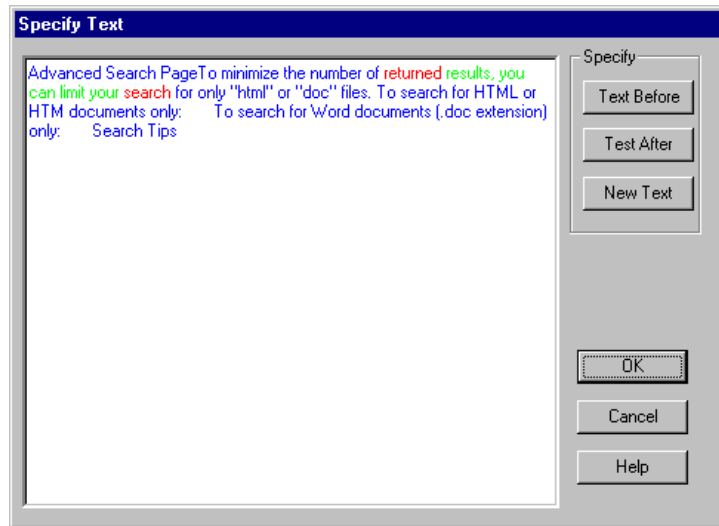
You can check whether a text string exists in an object or a frame using `web_obj_text_exists` or `web_frame_text_exists`.

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.



The text string to be checked is displayed in green and underlined. The bold 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.

11

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

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 219.

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 9, “Checking GUI Objects.”

At any time, you can view the current values of the properties of an ActiveX or a Visual Basic control using the 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 235. For information on checking the contents of an ActiveX table control, see Chapter 13, “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 Readme*.

Button Objects

The following ActiveX controls are supported for button objects:

- Infragistics (Sheridan) ActiveThreeD Control
- Infragistics (Sheridan) Data CommandButton Control
- Infragistics (Sheridan) OLE Data CommandButton Control

Calendar Objects

The following ActiveX controls are supported for calendar objects:

- Crescent CSCalendar Control
- Infragistics (Sheridan) MonthView Control

Check Box Objects

The following ActiveX controls are supported for check box objects:

- Infragistics (Sheridan) ActiveThreeD Control

Combo Box Objects

The following ActiveX controls are supported for combo box objects:

- Infragistics (Sheridan) Data Combo Control
- Infragistics (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
- Infragistics (Sheridan) ActiveToolBars Control
Infragistics (Sheridan) ActiveToolBars Plus Control

Radio Button Objects

The following ActiveX controls are supported for radio button objects:

- Infragistics (Sheridan) ActiveThreeD Control

Radio Group Objects

The following ActiveX controls are supported for radio group objects:

- Infragistics (Sheridan) Data Option Set Control
Infragistics (Sheridan) OLE Data Option Set Control

Tab Objects

The following ActiveX controls are supported for tab objects:

- Microsoft TabStrip Control
- Infragistics (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
- ▶ Infragistics (Sheridan) Data Grid Control
Infragistics (Sheridan) OLE DBGrid
Infragistics (Sheridan) DBData Option Set
Infragistics (Sheridan) OLEDBData Option Set
Infragistics (Sheridan) DBCombo
Infragistics (Sheridan) OLE DBCombo
Infragistics (Sheridan) DBData Command
Infragistics (Sheridan) OLEDBData Command

Toolbar Objects

The following ActiveX controls are supported for tool bar objects:

- ▶ DataDynamics ActiveBar Control
- ▶ Microsoft Toolbar Control
- ▶ Infragistics (Sheridan) ActiveToolBars Control
Infragistics (Sheridan) ActiveToolBars Plus Control

Tree Objects

The following ActiveX controls are supported for tree objects:

- ▶ Microsoft TreeView Control
- ▶ Infragistics (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
- ▶ use the `ActiveX_activate_method` TSL function to activate methods in the ActiveX control.

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 the *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 the *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.

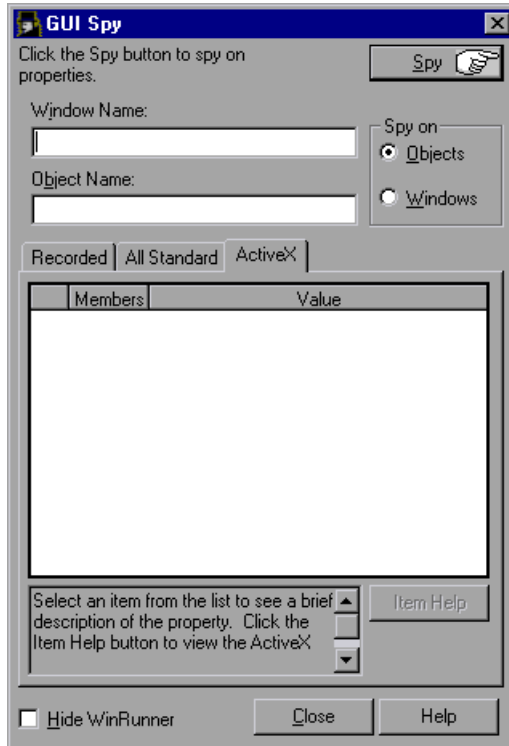
Viewing ActiveX and Visual Basic Control Properties

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 9, “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.

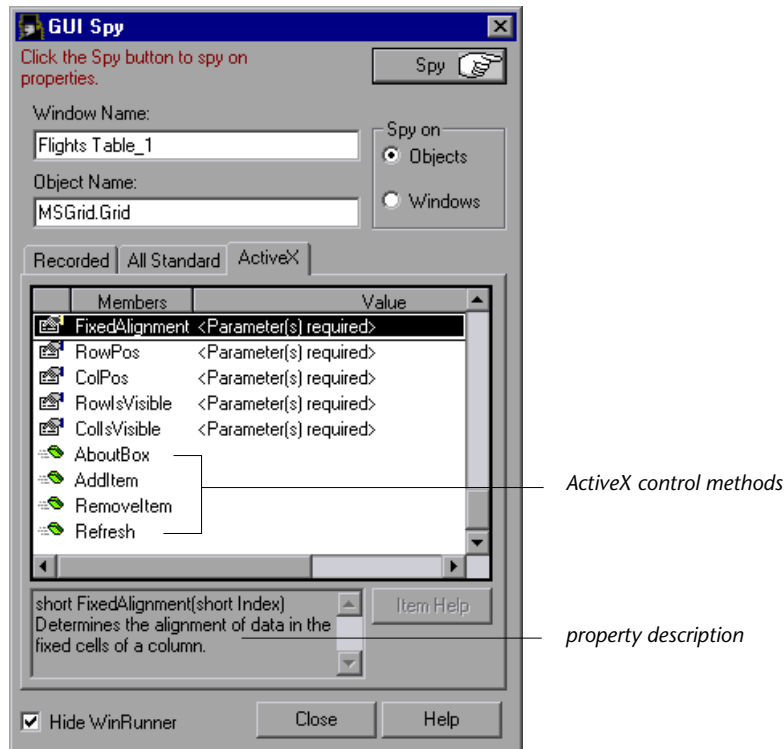


3 Click **Spy** and point to an ActiveX or Visual Basic control.

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:



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 226 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, refer to Chapter 8, "Generating Functions" in the *Mercury WinRunner Advanced Features User's Guide*.

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 223.

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.

Notes:

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 230.

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.

Notes:

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 230.

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

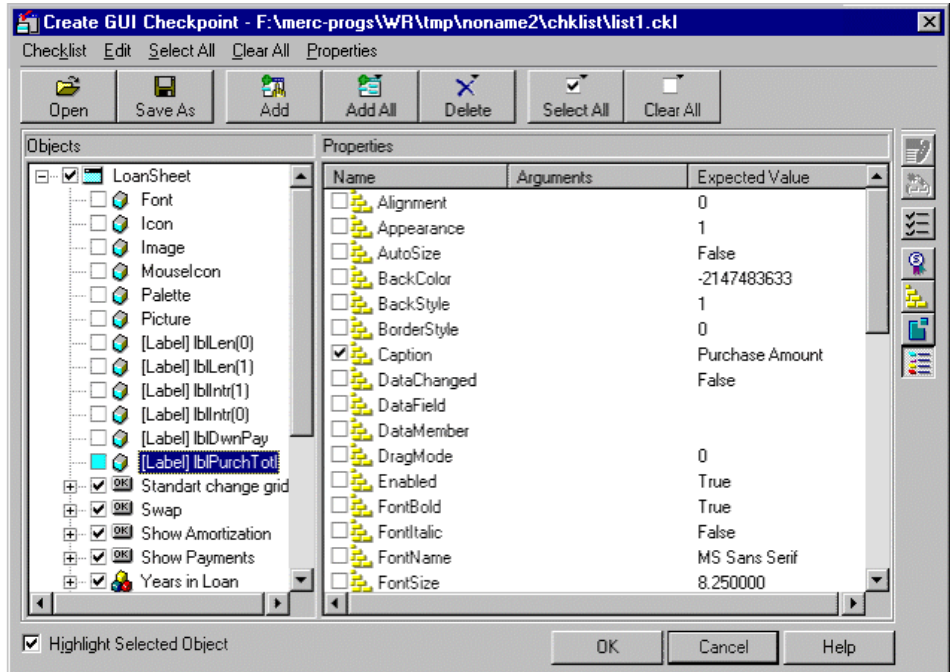
Creating GUI Checkpoints

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.



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 226.

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 226.

Checking Sub-Objects of ActiveX and Visual Basic Controls

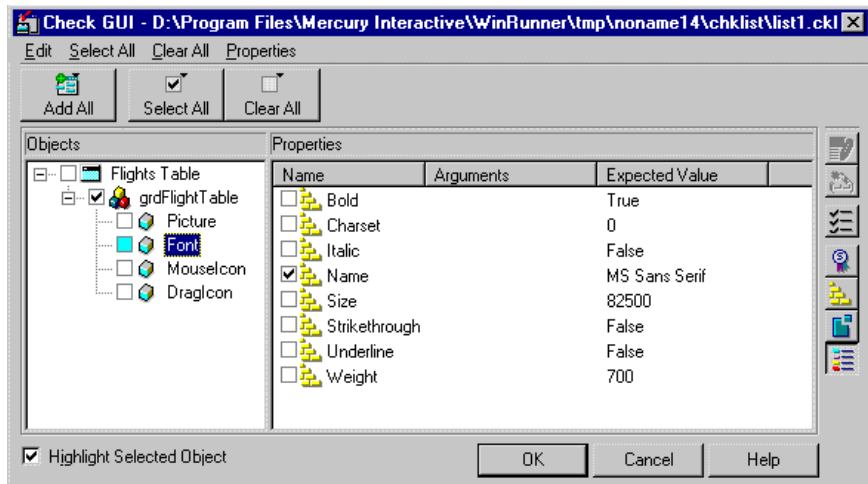
ActiveX and Visual Basic controls may contain sub-objects, which contain their own properties. An example of a sub-object is `Font`. Note that `Font` is a sub-object because it cannot be highlighted in the application you are testing. When you load the appropriate add-in support, you can create a GUI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

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

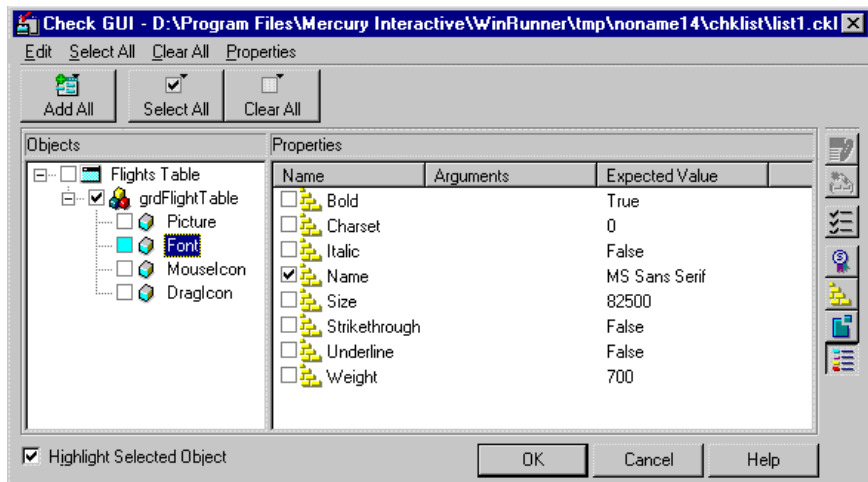
To check the sub-objects of an ActiveX or a Visual Basic control:



- 1 Choose **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 and then the Check GUI dialog box opens.



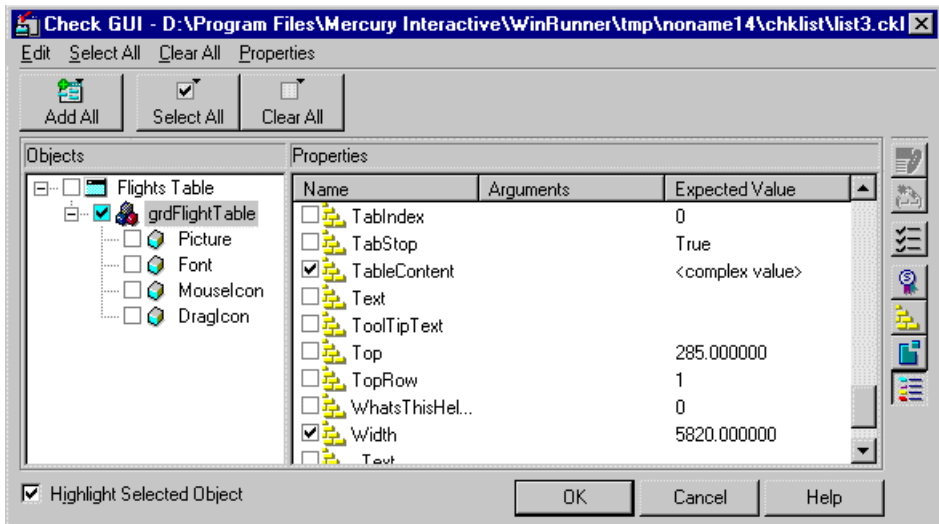
- 3 In the **Objects** pane, click the Expand sign (+) beside the object to display its sub-objects, and select a sub-object to display its ActiveX control properties.



The **Objects** pane displays the object and its sub-objects. In this example, the sub-objects are displayed under the “grdFlightTable” object. The **Properties** pane displays the properties of the sub-object that is highlighted in the Objects pane. Note that each sub-object has one or more default property checks. In this example, the properties of the Font sub-object are displayed, and the Name property of the Font sub-object is selected as a default check.

Specify which sub-objects of the table to check: first, select a sub-object in the Objects pane; next, select the properties to check in the Properties pane.

Note that since this ActiveX control is a table, by default, checks are selected on the **Height**, **Width**, and **TableContent** properties. If you do not want to perform these checks, clear the appropriate check boxes. For information on checking table contents, see Chapter 13, “Checking Table Contents.”



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

An `obj_check_gui` statement is inserted into your test script. For more information on the `obj_check_gui` function, see Chapter 9, “Checking GUI Objects,” or refer to the *TSL 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	Infragistics (Sheridan) Data Grid Control	Apex True DBGrid Control	Infragistics UltraGrid Control
<code>tbl_activate_cell</code>	+	+	+	+	+	+
<code>tbl_activate_header</code>	+	+	+	+	+	+
<code>tbl_get_cell_data</code>	+	+	+	+	+	+
<code>tbl_get_cols_count</code>	+	+	+	+	+	+
<code>tbl_get_column_name</code>	+	+	+	+	+	+
<code>tbl_get_rows_count</code>		+	+	+	+	+
<code>tbl_get_selected_cell</code>	+	+	+	+	+	+
<code>tbl_get_selected_row</code>	+	+		+	+	+
<code>tbl_select_col_header</code>	+	+	+	+	+	+
<code>tbl_set_cell_data</code>	+	+	+	+	+	+
<code>tbl_set_selected_cell</code>	+	+	+	+	+	+
<code>tbl_set_selected_row</code>	+	+	+		+	+

12

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 13, “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” below.

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



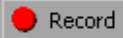
- 1** Choose **Test > Record–Context Sensitive** or click the **Record–Context Sensitive** button.
- 2** Click in the DropDown object to record a `tbl_set_selected_cell` statement in your test script.
- 3** While recording, press the CHECK GUI FOR OBJECT/WINDOW softkey.
- 4** Click in the DropDown object 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

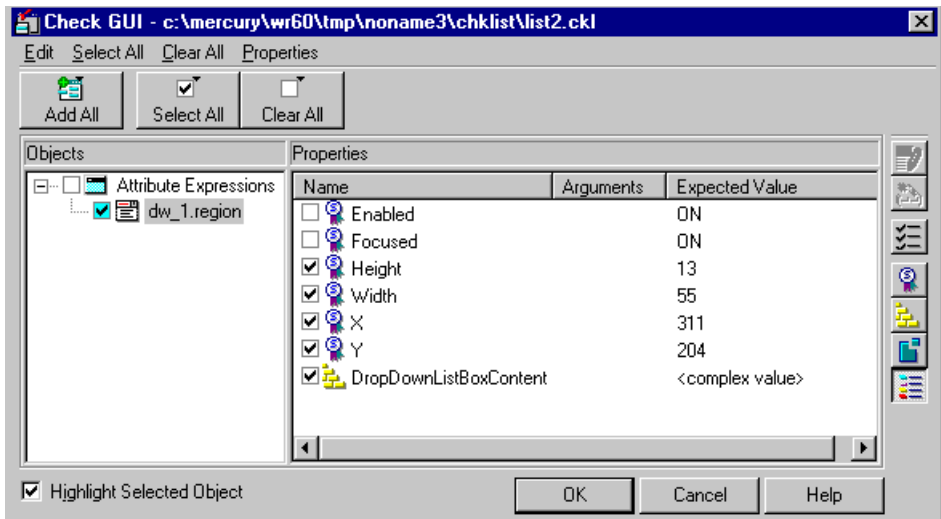
You can create a GUI checkpoint in which you specify which checks to perform on a DropDown object. When you double-click in a DropDown object while creating a GUI checkpoint, the Check GUI dialog box opens. If you are checking, for example, a DropDownListBox, you can double-click the `DropDownListBoxContent` 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.



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 253.
- 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 9, “Checking GUI Objects.”

Checking Properties of DataWindows

You can create a GUI checkpoint that checks the properties of a DataWindow. One of the properties you can check is **DWTableContent**, which is a check on the contents of the DataWindow. You create a content check on a DataWindow as you would create one on a table. For additional information on checking table contents, see Chapter 13, “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*.

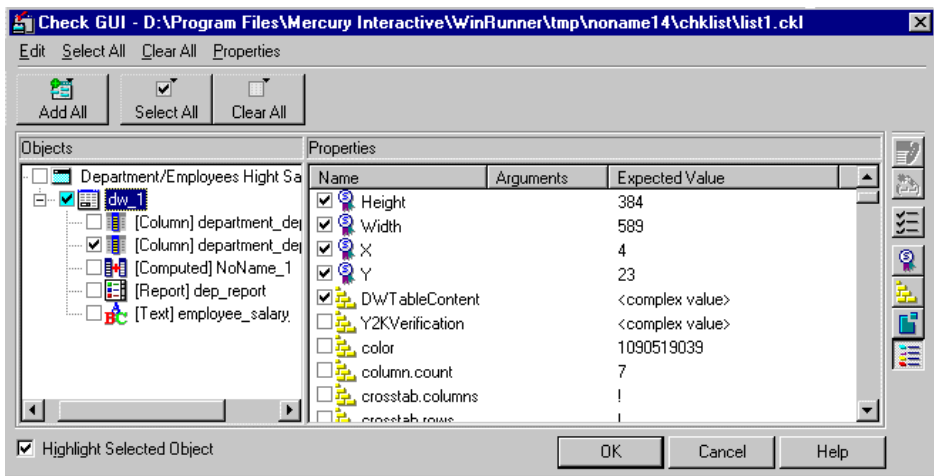
Checking Properties of a DataWindow while Specifying which Checks to Perform

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. The Check GUI dialog box opens.



Note that the properties of objects within a DataWindow are displayed in the dialog box. WinRunner can perform checks on these objects. For additional information, see “Checking Properties of Objects within DataWindows” below.



- 3** 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.
- 4** 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 253.
- 5** When you are done, click **OK** to save your changes, close the Edit Check dialog box, and restore the Check GUI dialog box.
- 6** 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:

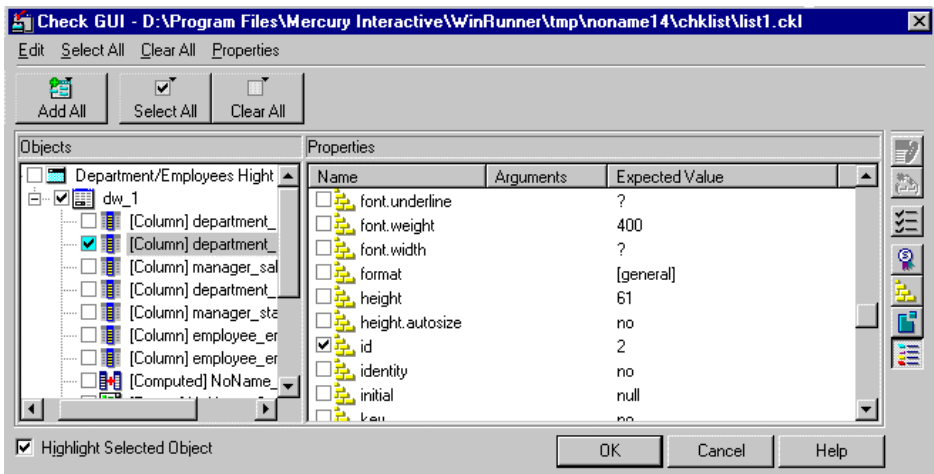
- ▶ DataWindows
- ▶ DataWindow columns
- ▶ DataWindow text
- ▶ DataWindow reports
- ▶ DataWindow graphs
- ▶ DataWindow computed columns

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 9, “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 and then the Check GUI dialog box opens.
- 3 In the **Objects** pane, click the Expand sign (+) beside the DataWindow to display its objects, and select an object to display its properties.



The **Objects** pane displays the DataWindow and the objects within it. The **Properties** pane displays the properties of the object in the DataWindow that is highlighted in the Objects pane. These objects can be columns, computed columns, text, graphs, and reports. Note that each object has one or more default property checks.

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

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

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

13

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 23, “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 9, “Checking GUI Objects.”

When you run the test, WinRunner compares the current state of the properties in the table to the expected results. If the expected results and the current results do not match, the GUI checkpoint fails. You can view the results of the checkpoint in the WinRunner Test Results Window. For more information, see Chapter 21, “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 12, “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 11, “Working with ActiveX and Visual Basic Controls.” You can also check a table’s standard GUI properties in a GUI checkpoint. For additional information, see Chapter 9, “Checking GUI Objects.”

Checking Table Contents with Default Checks

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

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

You can also perform a check on table contents in which you specify which checks to perform. For additional information, see “Checking Table Contents while Specifying Checks” on page 250.

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 9, “Checking GUI Objects.” For information on checking the ActiveX control properties of a tables, see Chapter 11, “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.

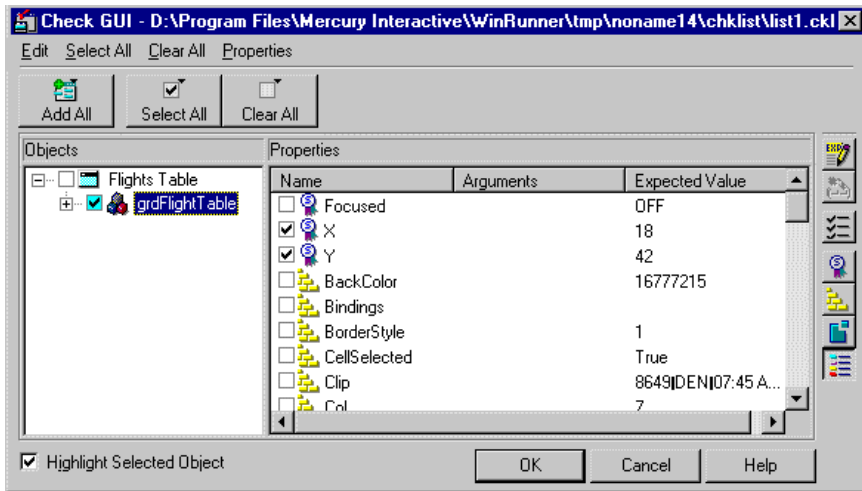
The example in the procedure below uses WinRunner with add-in support for Visual Basic and the Flights table in the sample Visual Basic Flights application.

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.

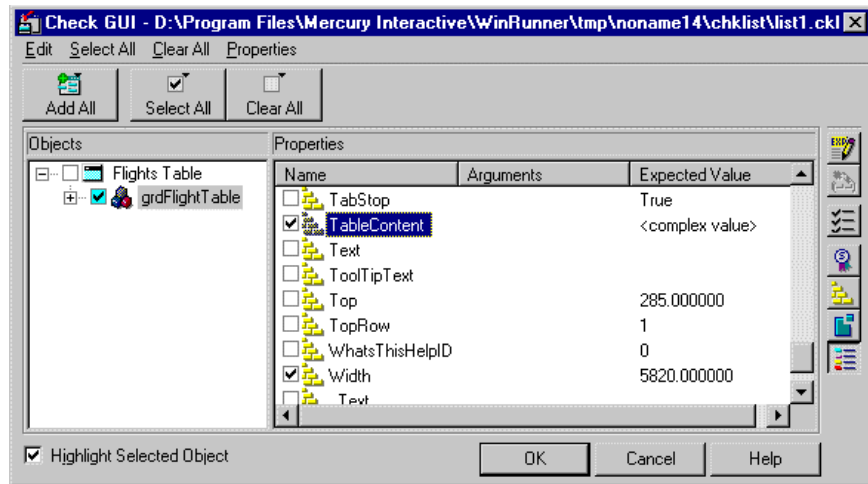
WinRunner may take a few seconds to capture information about the table, and then the Check GUI dialog box opens.



The dialog box displays the table's unique table properties as nonstandard objects.

- 3 Scroll down in the dialog box or resize it so that the **TableContent** property check is displayed in the **Properties** pane.

Note that the table contents property check may have a different name than **TableContent**, depending on which toolkit is used.



- 4 Select the **TableContent** (or corresponding) property check and click the **Edit Expected Value** button. Note that <complex value> appears in the Expected Value column for this property check, since the expected value of this check is too complex to be displayed in this column.

The Edit Check dialog box opens.

- 5 You can select which cells to check and edit the expected data. For additional information on using this dialog box, see “Understanding the Edit Check Dialog Box” on page 253.
- 6 When you are done, click **OK** to save your changes, close the Edit Check dialog box, and restore the Check GUI dialog box.

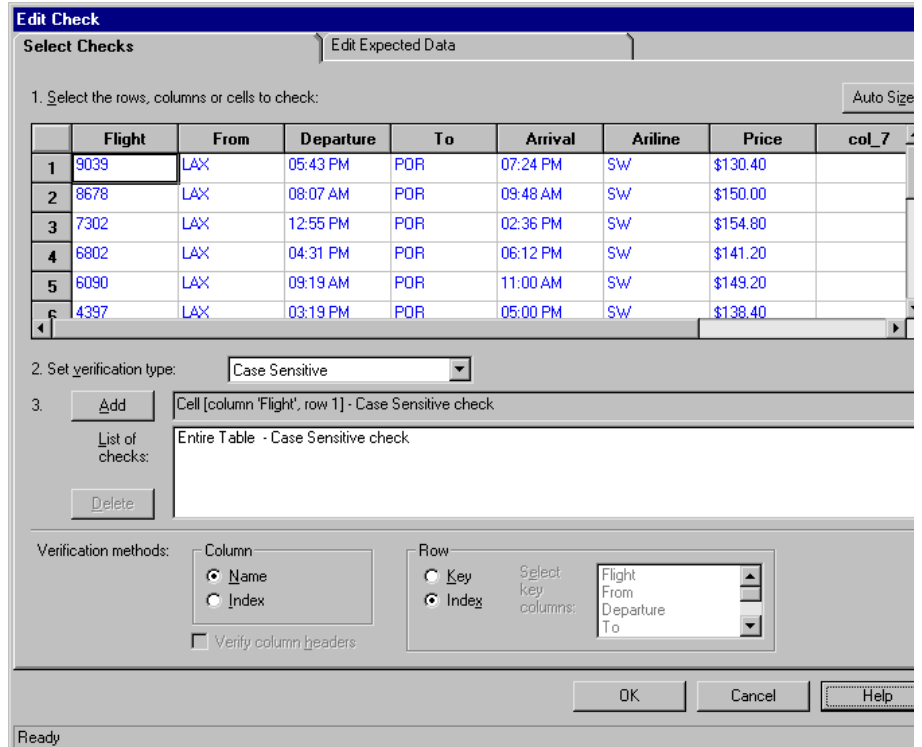
- 7 Click **OK** to close the Check GUI dialog box.

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 9, “Checking GUI Objects.” For information on checking the ActiveX control properties of a tables, see Chapter 11, “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.



In the **Select Checks** tab, you can specify which table cells to check, the verification method, and 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 more information, see “Specifying the Verification Method for a Single-Column Table” on page 256.

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 257.

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:

- 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 254. Choosing this option enables the **Verify column headers** check box, which enables you to check column headers as well as cells.

Row

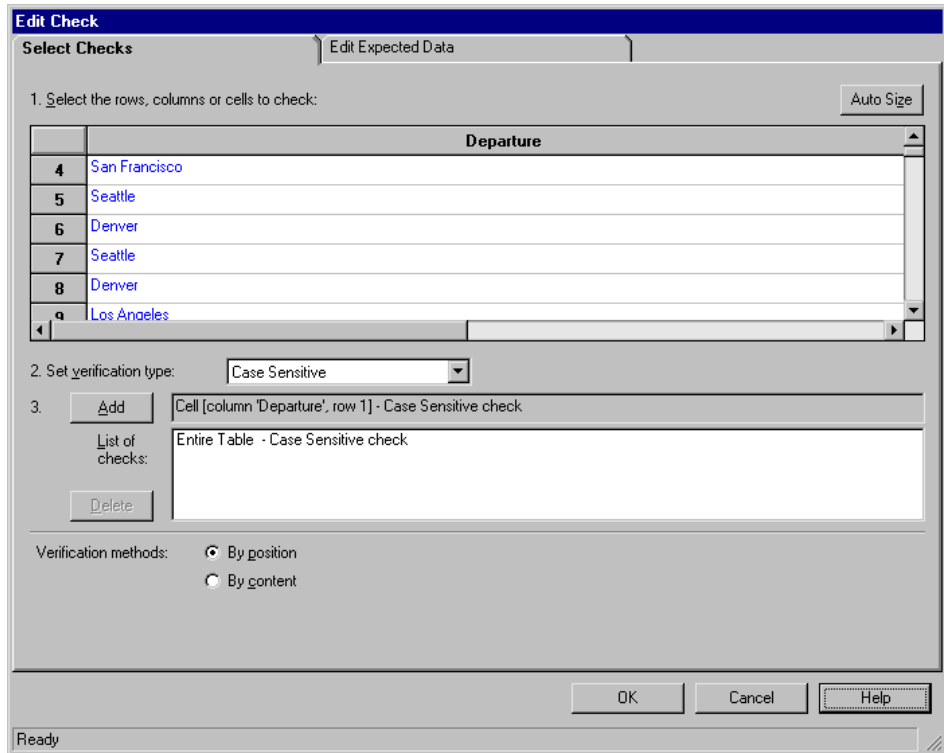
- **Key:** WinRunner looks for the rows in the selection according to the data in the key column(s) specified in the **Select key columns** list box. For example, you could tell WinRunner to identify the second row in the table on page 258 based on the arrival time for that row. A shift in the position 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.



- **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	Airline	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.

14

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
- ▶ Creating a Runtime Database Record Checkpoint
- ▶ Editing a Runtime Database Record Checklist
- ▶ Creating a Default Check on a Database
- ▶ Creating a Custom Check on a Database
- ▶ Messages in the Database Checkpoint Dialog Boxes
- ▶ Working with the Database Checkpoint Wizard
- ▶ Understanding the Edit Check Dialog Box
- ▶ Modifying a Standard Database Checkpoint
- ▶ Modifying the Expected Results of a Standard Database Checkpoint
- ▶ Parameterizing Standard Database Checkpoints
- ▶ Specifying a Database
- ▶ Using TSL Functions to Work with a Database

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. You 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 21, “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 269 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. You 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 21, “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. For more information, see “Parameterizing Standard Database Checkpoints,” on page 310.

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 579.

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 562.

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 269 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.

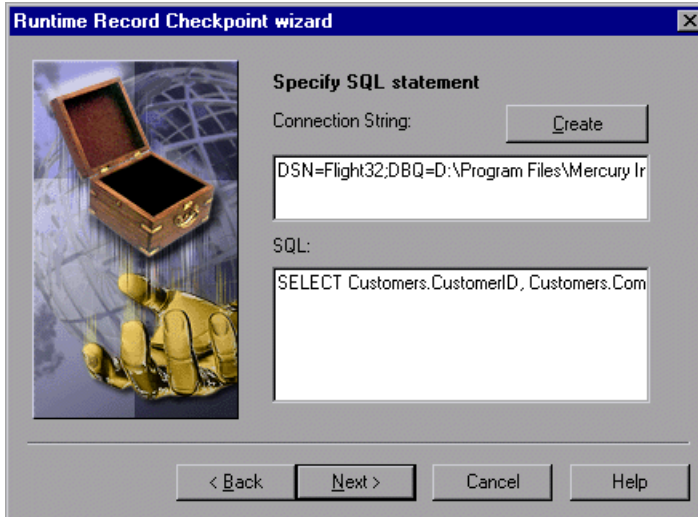


You can choose from the following options:

- **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 314. 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 287.

Specify SQL Statement Screen

The Specify SQL Statement screen enables you to manually specify the database connection string and the SQL statement.



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.)



- **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.)



- **Select text from a Web page:** Enables you to indicate the text on your Web page containing the value to be verified.

Notes:

When selecting text from a Web page, you must use the pointer to select the text.

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.



- **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*.

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 264, 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 270.

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.

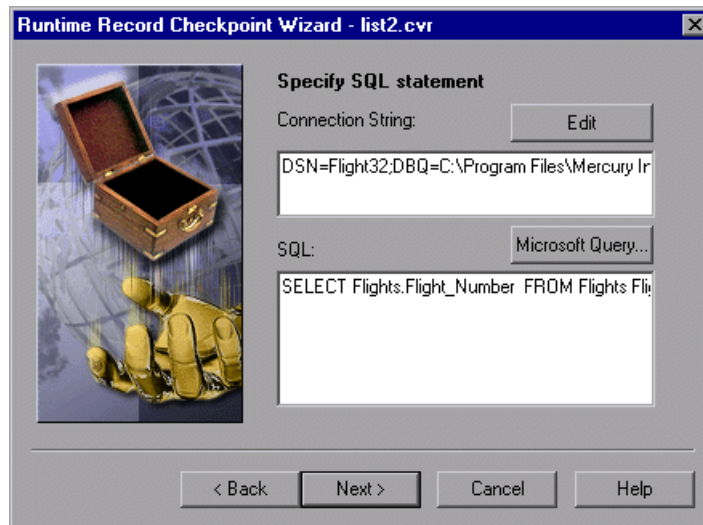


- 2 Choose the runtime database record checklist to edit. Click **Next** to proceed.

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.

- 3 The Specify SQL statement screen opens:



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 following screen opens:



“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.



- 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 267.

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 280.

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 283.

- 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 284.

- 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 314 or “Specifying an SQL Statement” on page 287.

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 313.

- 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 283.

- 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 315.
- 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 283.
- 3** If you are creating a new query, an instruction screen opens for creating a query.

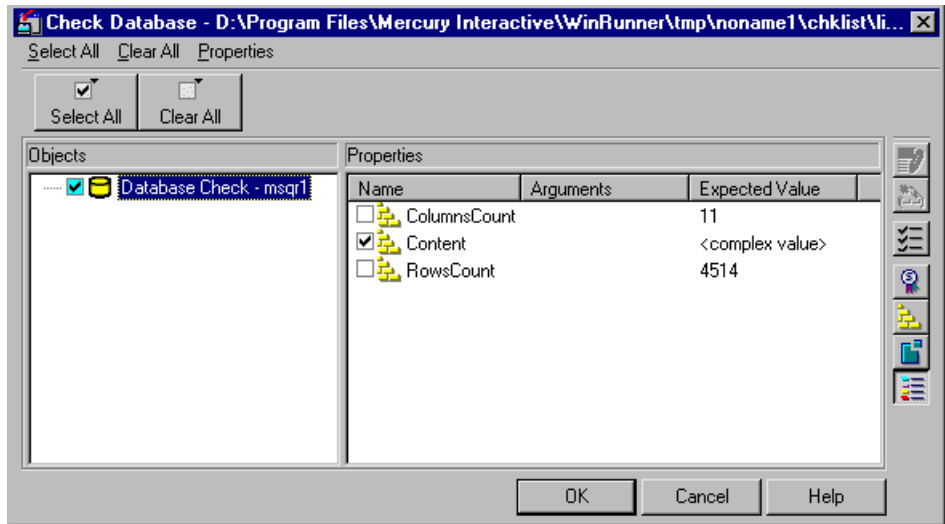
If you do not want to see this message next time you create a default database checkpoint, clear the **Show this message next time** check box.

- 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 314 or “Specifying an SQL Statement” on page 287.

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 315.

- 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 310.
- 6 WinRunner takes several seconds to capture the database query and restore the WinRunner window.

The Check Database dialog box opens.



The **Objects** pane contains “Database check” and the name of the **.sql* query file 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 277.

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 **RowsCount** 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 291.

- 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 21, “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

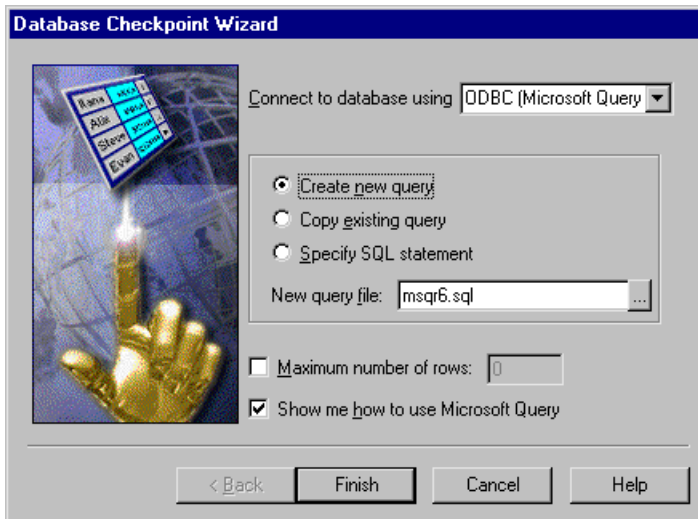
ODBC (Microsoft Query) Screens

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.



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 280.
- ▶ **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 286.
- ▶ **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 287.
- ▶ **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.



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 280.

Specifying an SQL Statement

The following screen opens if you chose to specify an SQL statement to use in this database checkpoint.



In this screen you must specify the connection string and the SQL statement:

- **Connection String:** Enter the connection string, or click **Create** to open the ODBC Select Data Source dialog box, in which you can select a *.dsn file, which inserts the connection string in the box.
- **SQL:** Enter the SQL statement.

Note: If you create an SQL statement containing parameters, an instruction screen opens. For information on parameterizing SQL statements, see "Parameterizing Standard Database Checkpoints" on page 310.

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 280.

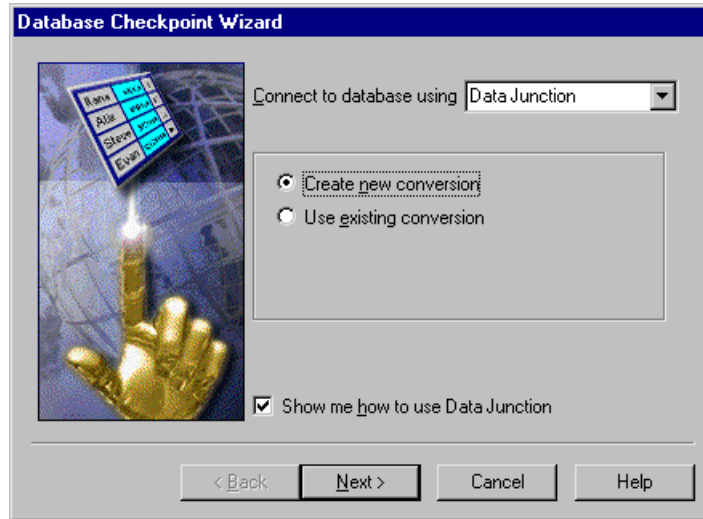
Data Junction Screens in the Database Checkpoint Wizard

There are two screens in the Database Checkpoint wizard for working with Data Junction. These screens enable you to:

- set general options:
 - switch to ODBC (Microsoft Query) mode
 - choose to create a new conversion or use an existing one
 - display an instruction screen
- specify the conversion file

Setting Data Junction Options

The following screen opens if you last worked with Data Junction or if you are creating a default database checkpoint for the first time when only Data Junction is installed:

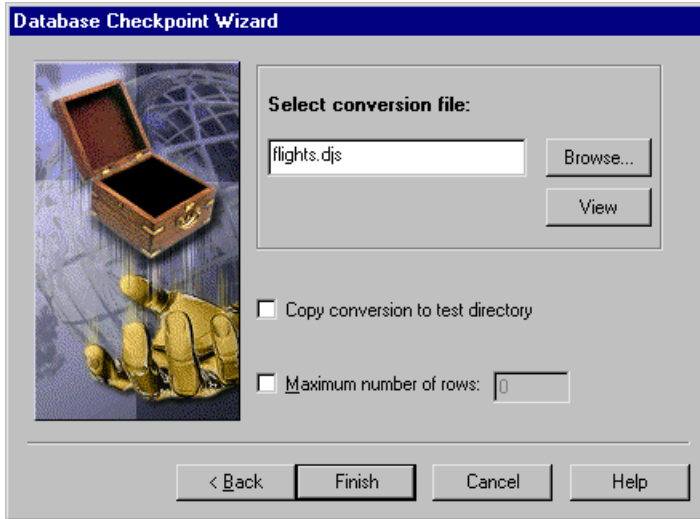


You can choose from the following options:

- **Create new conversion:** Opens Data Junction and enables you to create a new conversion file. For additional information, see “Creating a Conversion File in Data Junction” on page 315. 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 290.
- **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 290.
- **Show me how to use Data Junction** (available only when **Create new conversion** is selected): Displays instructions for working with Data Junction.

Selecting a Data Junction Conversion File

The following wizard screen opens when you are working with Data Junction.



Enter the pathname of the conversion file or use the **Browse** button to locate it. Once a conversion file is selected, you can use the **View** button to open the file for viewing.

You can also choose from the following options:

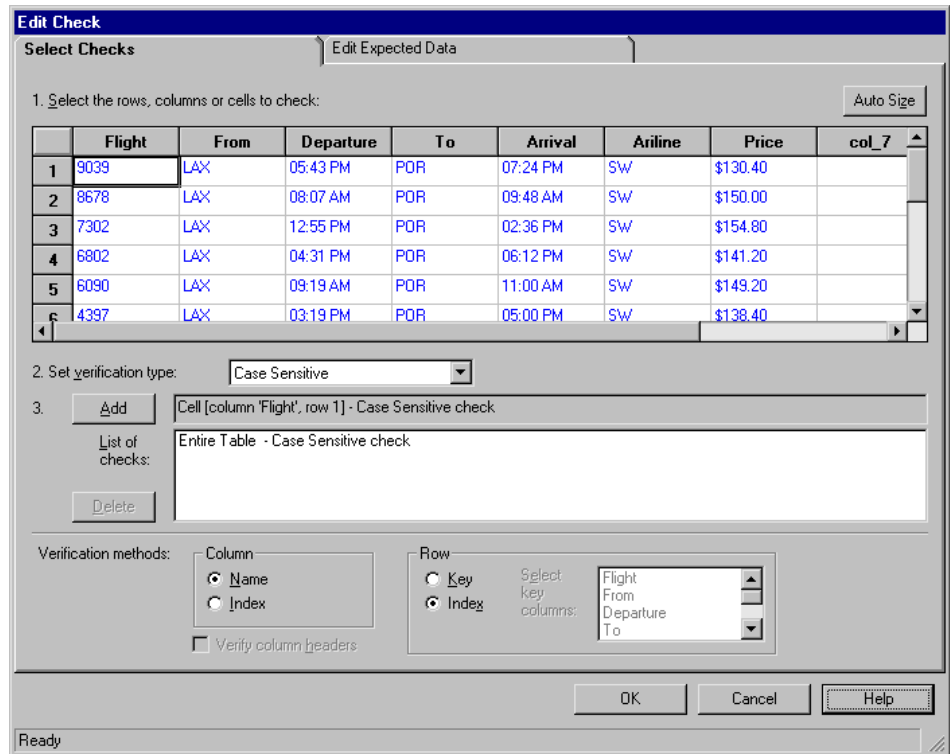
- **Copy conversion to test folder:** Copies the specified conversion file to the test folder.
- **Maximum number of rows:** Select this check box and enter the maximum number of database rows to check. If this check box is cleared, there is no maximum.

When you are done:

- If you are creating a default database checkpoint, a **db_check** statement is inserted into your test script.
- If you are creating a custom database checkpoint, the Check Database dialog box opens. For information on the Check Database dialog box, see “Creating a Custom Check on a Database” on page 280.

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 280.)



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 294.

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 295.

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:

- ▶ 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

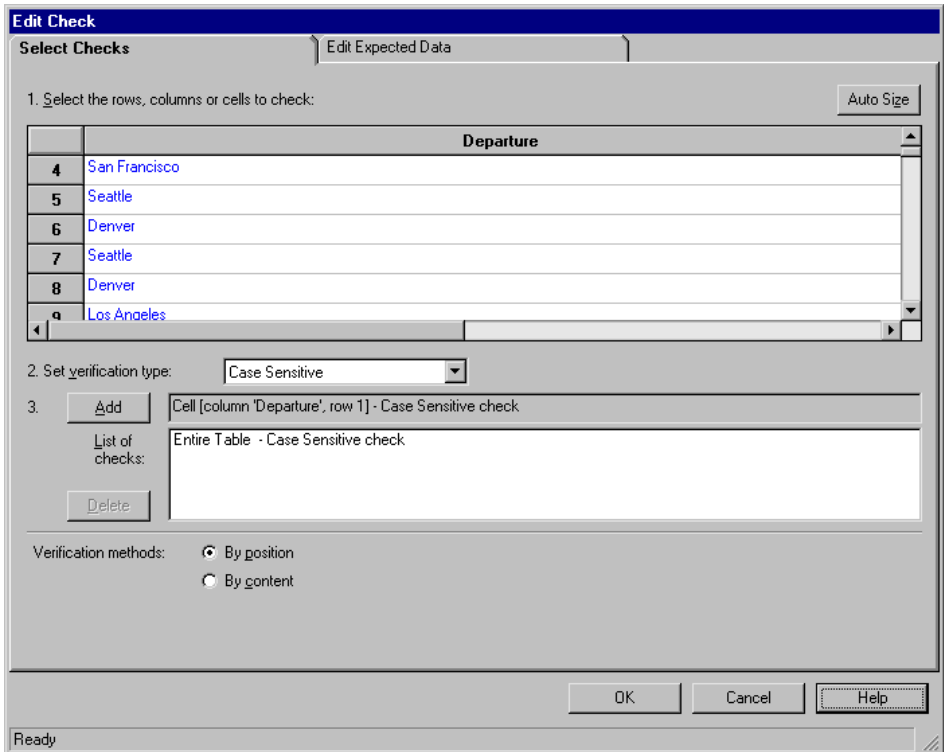
- ▶ **Name:** (default setting): WinRunner looks for the selection according to the column names. A shift in the position of the columns within the result set does not result in a mismatch.
- ▶ **Index:** WinRunner looks for the selection according to the index, or position, of the columns. A shift in 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 292. 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.



- ▶ **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.

Edit Check								
Select Checks								
Edit Expected Data								
	Flight	From	Departure	To	Arrival	Airline	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	

Ready

OK Cancel Help

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 308.

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.

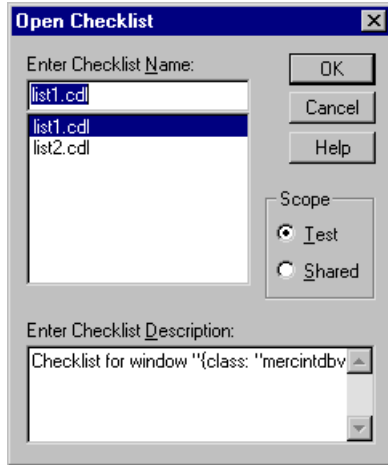
Note: *.sql files are not saved in shared database checklist folders.

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 23, “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.

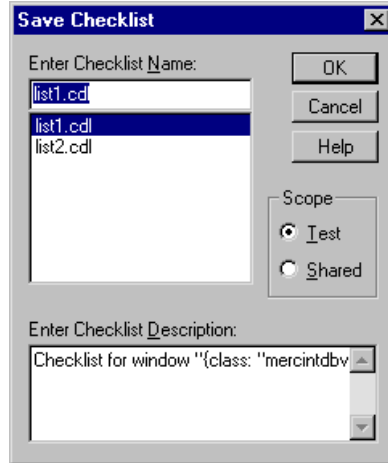


- 2 Select a database checklist and click **OK**. Note that database checklists have the **.cdl** extension, while GUI checklists have the **.ckl** extension. For information on GUI checklists, see “Modifying GUI Checklists,” on page 143.

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

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

The Save Checklist dialog box opens.



- 4 Under **Scope**, click **Shared**. Type in a name for the shared checklist. Click **OK** to save the checklist and close the dialog box.
- 5 Click **OK** to close the Edit Database Checklist dialog box.

Editing Database Checklists

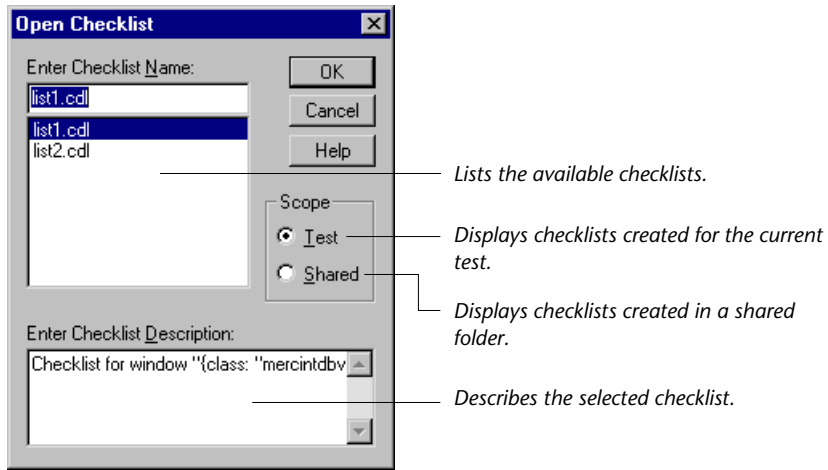
You can edit an existing database checklist. Note that a database checklist includes only a reference to the *msqr*.sql* query file or the **.djs* conversion file of the database and the properties to be checked. It does not include the expected results for the values of those properties.

You may want to edit a database checklist to change which properties in a database to check.

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 297.

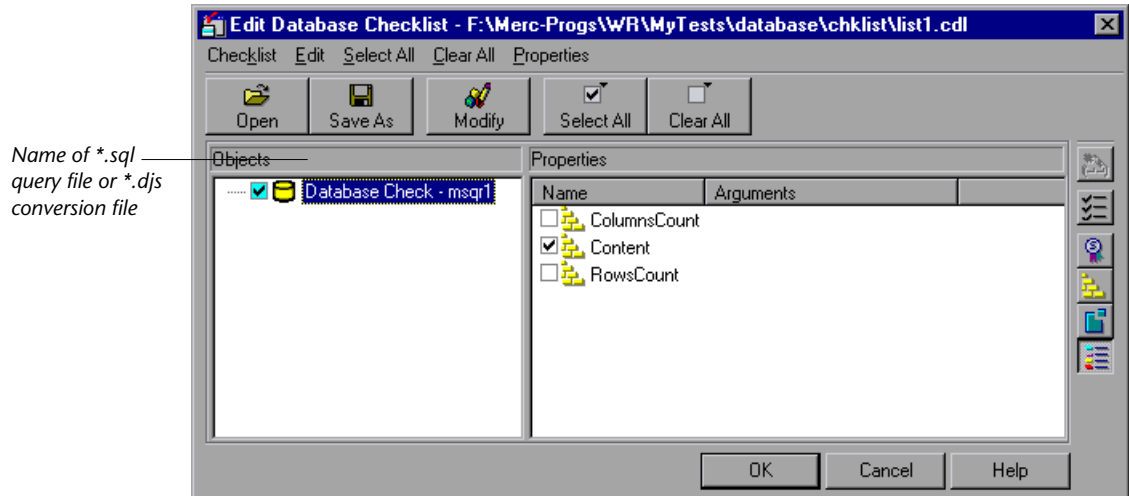


3 Select a database checklist.

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.



You can use the **Modify** button to modify the database checkpoint, as described in “Modifying a Query in an Existing Database Checklist” on page 302.

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 277.

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 429.

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.

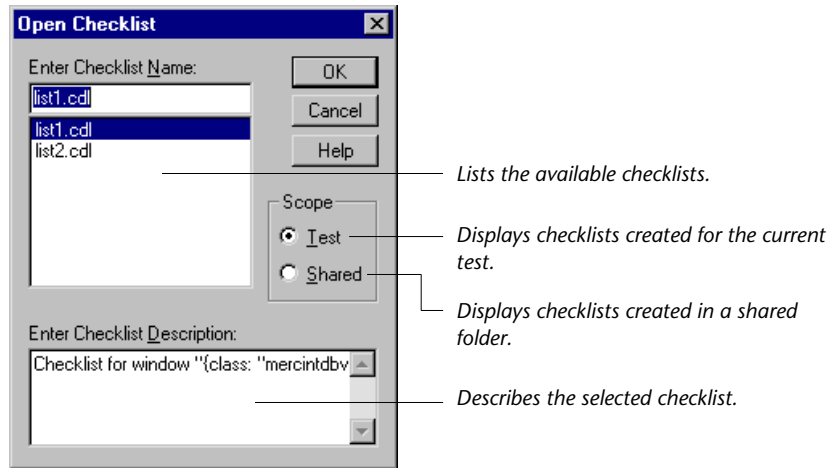
Modifying a Query Created with ODBC/Microsoft Query

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 297.



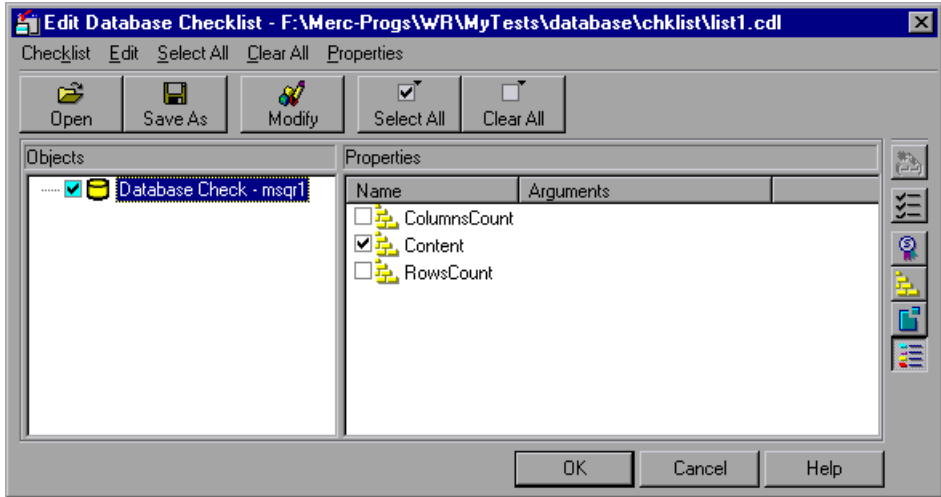
3 Select a database checklist.

4 Click **OK**.

The Open Checklist dialog box closes. The Edit Database Checklist dialog box opens and displays the selected checklist.

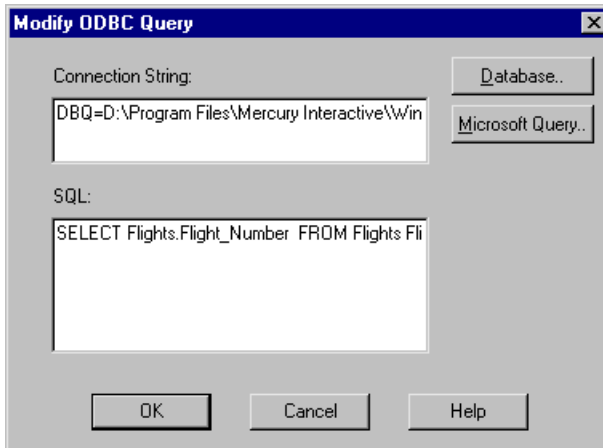
The **Objects** pane contains “Database check” and the name of the *.sql query file that will be included in the database checkpoint.

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 299.



- 5 In the **Objects** column, highlight the name of the query file or the conversion file, and click **Modify**.

The Modify ODBC Query dialog box opens.



- 6 Modify the ODBC query by changing the connection string and/or the SQL statement. You can click **Database** to open the ODBC Select Data Source dialog box, in which you can select a *.*dsn* file, which inserts the connection string in the box. You can click **Microsoft Query** to open Microsoft Query.
- 7 Click **OK** to return to the Edit Checklist dialog box.
- 8 Click **OK** to close the Edit Checklist dialog box.

Note: You must run all tests that use this checklist in Update mode before you run them in Verify mode. For more information, see “Running a Test to Update Expected Results” on page 438.

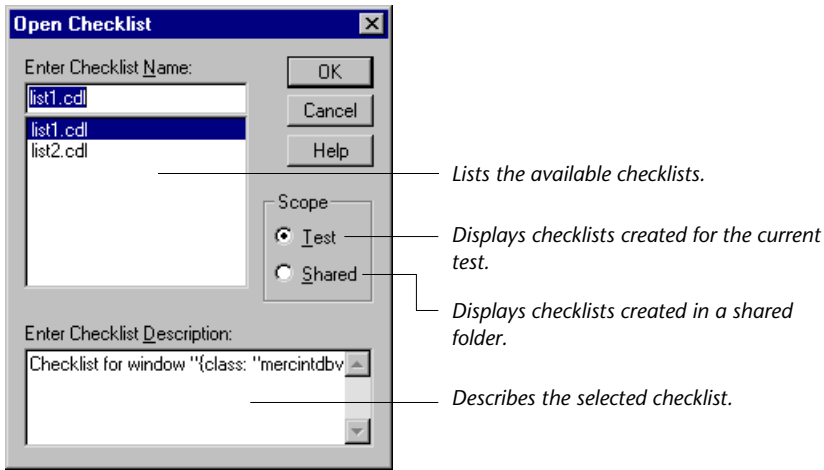
Modifying a Query Created with Data Junction

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 297.



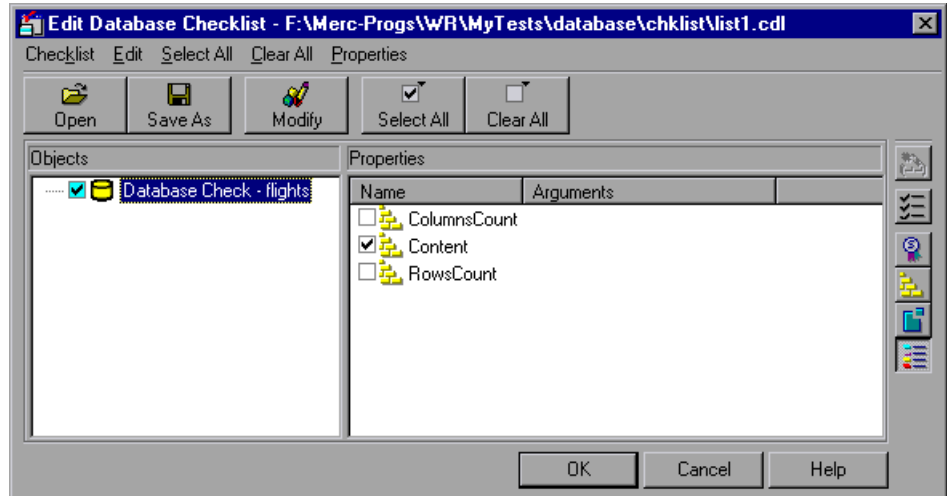
3 Select a database checklist.

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 299.



- 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 438.

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.

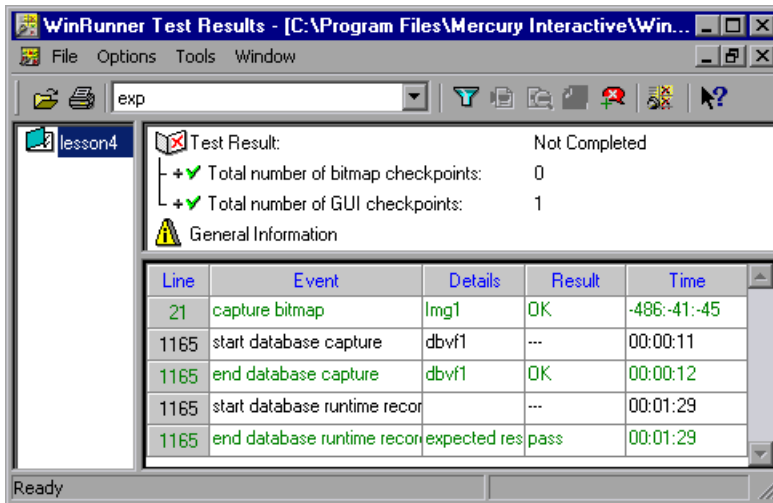
To modify the expected results for an existing database checkpoint:



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

The WinRunner Test Results window opens.

- 2 Select **exp** in the results location box.

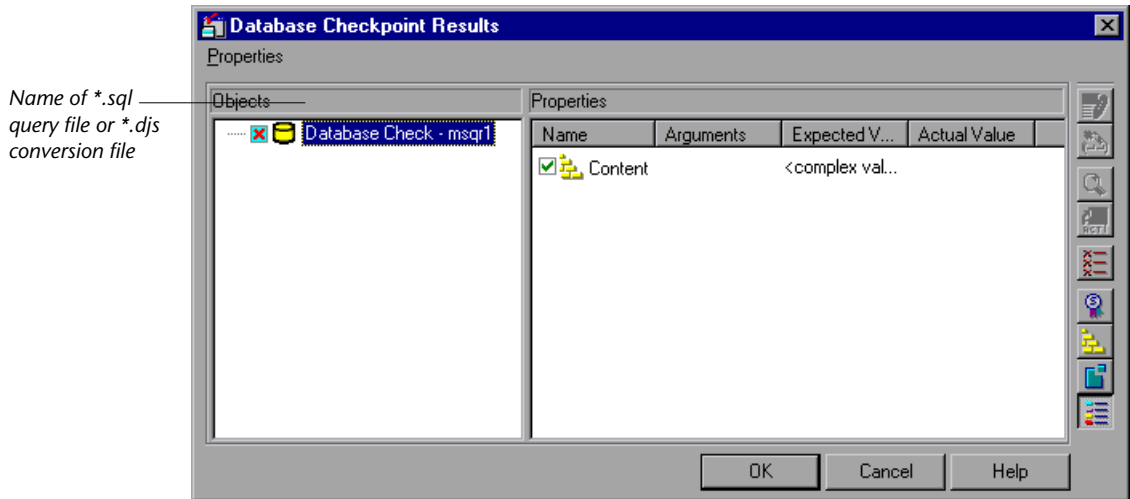


- 3 Locate the database checkpoint by looking at **end database capture** entries.



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.



- 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 database checkpoint. For more information, see “Creating a Custom Check on a Database” on page 280.

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 499.

For more information on working in the Test Results window, see Chapter 21, “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
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.

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 287.

- 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 280. 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 283.

- 3** A message box prompts you with instructions, which are also described below. Click **OK** to close the message box.

The WinRunner window is restored and a **db_check** statement similar to the following is inserted into your test script.

```
db_check("list1.cdl", "dbvf1", NO_LIMIT, dbvf1_params);
```

- 4** Create 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 310 with the new values. Follow the guidelines below for adding an array in TSL to parameterize your SQL statements.

- 5** Run your test in Update mode to update the SQL statement with these values.

After you have completed this procedure, you can run your test in Verify mode with the SQL statement. To change the parameters in the SQL statement, you modify the 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.

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 287.

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 313.

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 278.
- ▶ To create a custom check on a database, follow the instructions starting at step 6 on page 281.

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 290.
- 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 278.
 - To create a custom check on a database, follow the instructions starting at step 6 on page 281.

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, refer to Chapter 8, “Generating Functions” in the *Mercury WinRunner Advanced Features User’s Guide*. 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 277 and “Creating a Custom Check on a Database” on page 280. For information on parameterizing `db_check` statements, see “Parameterizing Standard Database Checkpoints” on page 310.

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 264.

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 on 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* is 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* is 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.

15

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.

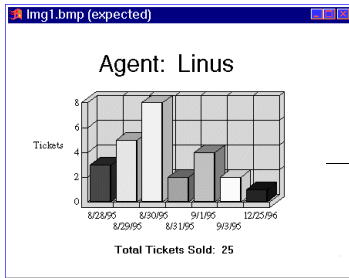
This chapter describes:

- ▶ About Checking Bitmaps
- ▶ Creating Bitmap Checkpoints
- ▶ Checking Window and Object Bitmaps
- ▶ Checking Area Bitmaps

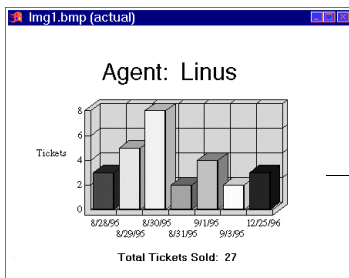
About Checking Bitmaps

You can check an object, a window, or an area of a screen in your application as a bitmap. While creating a test, you indicate what you want to check. WinRunner captures the specified bitmap, stores it in the expected results folder (*exp*) of the test, and inserts a checkpoint in the test script. When you run the test, WinRunner compares the bitmap currently displayed in the application being tested with the *expected* bitmap stored earlier. In the event of a mismatch, WinRunner captures the current *actual* bitmap and generates a *difference* bitmap. By comparing the three bitmaps (expected, actual, and difference), you can identify the nature of the discrepancy.

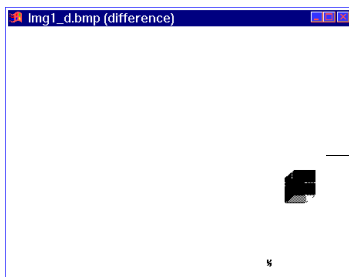
Suppose, for example, your application includes a graph that displays database statistics. You could capture a bitmap of the graph in order to compare it with a bitmap of the graph from a different version of your application. If there is a difference between the graph captured for expected results and 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.”

You 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 21, “Analyzing Test Results.”

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 397.

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 operating system folder, e.g. *C:\WINNT*.
- 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 579.

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 562.

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, refer to Chapter 21, “Setting Testing Options from a Test Script” in the *Mercury WinRunner Advanced Features User’s Guide*. 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 23, “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, refer to Chapter 21, “Setting Testing Options from a Test Script” in the *Mercury WinRunner Advanced Features User’s Guide*. 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 23, “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*.

16

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. If the Text Recognition Mechanism is set to driver based recognition, this process is automatic. However, if the Text Recognition Mechanism is set to image-based recognition, 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 342. For more information on setting the Text Recognition Mechanism, see “Setting Text Recognition Options” on page 558.

Note: When using the WinRunner text-recognition mechanism for Windows-based applications, keep in mind that it may occasionally retrieve unwanted text information (such as hidden text and shadowed text, which appears as multiple copies of the same string).

Additionally, the text recognition may behave differently in different run sessions depending on the operating system version you are using, service packs you have installed, other installed toolkits, the APIs used in your application, and so on.

Therefore, when possible, it is highly recommended to retrieve or check text from your application window by inserting a standard GUI checkpoint and selecting to check the object's **value** (or similar) property.

For additional details, see “Considerations for Using Text Recognition for Windows-Based Applications” on page 560.

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 recognize your text. To enable WinRunner to identify text, use the image-based text recognition mechanism and teach WinRunner your application fonts. For more information, see “Teaching Fonts to WinRunner” on page 342.

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 the Get Text dialog box 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 10, “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 342 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, refer to Chapter 8, “Generating Functions” in the *Mercury WinRunner Advanced Features User’s Guide*. 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 [ ,x1,y1,x2,y2 ] [ ,string_def ] );
```

```
obj_find_text ( object, string, result_array [ ,x1,y1,x2,y2 ] [ ,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 *x₁,y₁,x₂,y₂* 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, [ ,x1,y1,x2,y2] [ ,string_def ] );
```

```
obj_move_locator_text ( object, string, [ ,x1,y1,x2,y2] [ ,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, [ ,x1,y1,x2,y2 ] [ ,string_def ]
[ ,mouse_button ] );
```

The *window* or *object* is the window or object to search. The *string* is the text the mouse pointer clicks. The optional *x1,y1,x2,y2* parameters specify the region of the window or object that is searched. The optional *string_def* defines how WinRunner searches for the text. The optional *mouse_button* specifies which mouse button to use.

In the following example, **obj_click_on_text** clicks a topic in an online help index in order to jump to a help topic.

```
function show_topic(win,str)
{
    auto text,text1,rc,arr[];

    # Search for the topic string within the object. If not found, scroll down to end
    # of document.
    set_window (win, 1);
    obj_mouse_click ("MS_WINTOPIC", 1, 1, LEFT);
    type("<kCtrl_L-kHome_E>");
    while(rc=obj_click_on_text("MS_WINTOPIC",str,TRUE,LEFT)){
        type("<kPgDn_E>");
        obj_get_text("MS_WINTOPIC", text);
        if(text==text1)
            return E_GENERAL_ERROR;
        text1=text;
    }
}
```

For information about the **click_on_text** functions, refer to the *TSL 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.

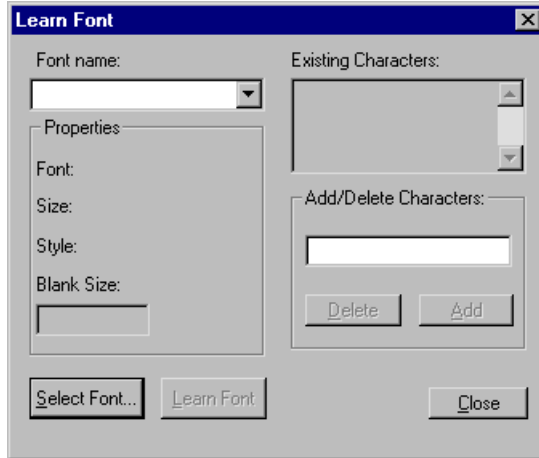
Learning a Font

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**.

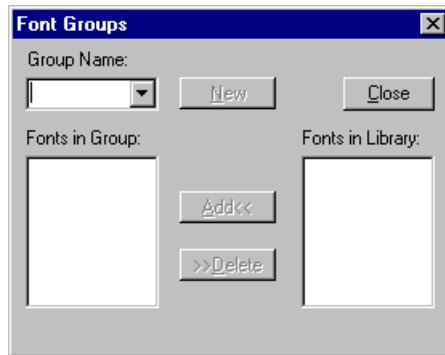
Creating a Font Group

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:

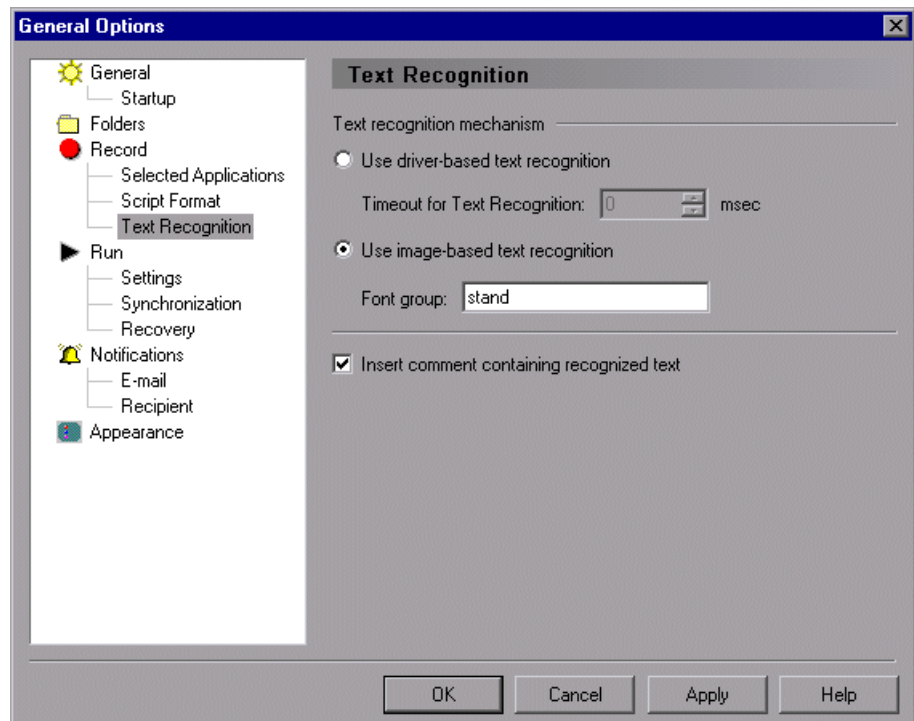
- 1** In the Fonts Expert window, choose **Font > Groups**. The Font Groups dialog box opens.
- 2** Select the desired font group from the **Group Name** list.
- 3** In the **Fonts in Group** list, click the name of the font to delete.
- 4** Click **Delete**.

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.



- 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 23, “Setting Global Testing Options.” For more information about using the `setvar` function to choose a font group from within a test script, refer to Chapter 21, “Setting Testing Options from a Test Script” in the *Mercury WinRunner Advanced Features User’s Guide*.

17

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 23, "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 352 and "Using an Existing Date Format Configuration File" on page 354.
- 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 355.
- 3** Run the tests (in Debug mode) to check that they run smoothly. For more information, see Chapter 20, "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 358.

- 4** Run the tests (in Update mode) to create expected results. For more information, see Chapter 20, "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 20, “Understanding Test Runs.”
- 6 Analyze test results to pinpoint where date-related problems exist in the application. For more information, see Chapter 21, “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 351. For information on date formats, see “Setting Date Formats” on page 352 and “Using an Existing Date Format Configuration File” on page 354.

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

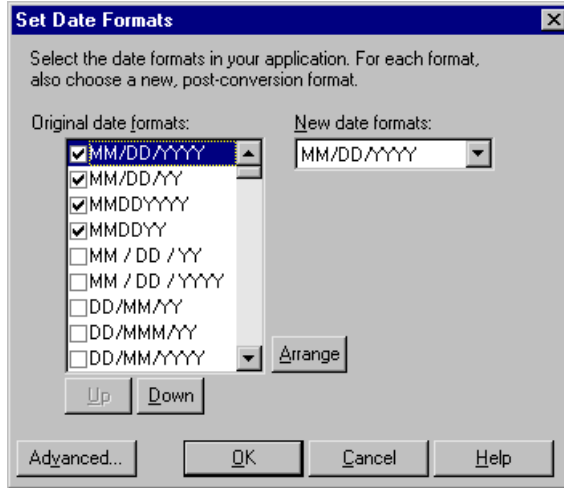
WinRunner supports a wide range of date formats. Before you begin creating tests, you should specify the date formats currently used in your application. This enables WinRunner to recognize date information when you insert checkpoints into a test script and run tests.

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.



- 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 354.

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
<i>y2k_expn.eur</i>	<ul style="list-style-type: none"> • 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).
<i>y2k_expn.us</i>	<ul style="list-style-type: none"> • 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).
<i>y2k_wind.eur</i>	<ul style="list-style-type: none"> • 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).
<i>y2k_wind.us</i>	<ul style="list-style-type: none"> • 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 9, "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 9, "Checking GUI Objects," and Chapter 13, "Checking Table Contents."

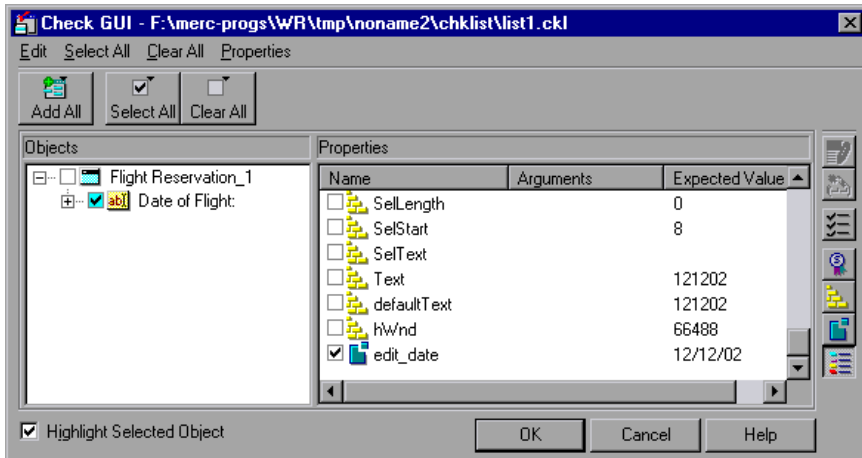
Checking Dates Using the Check GUI Dialog Box

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.



- 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 9, “Checking GUI Objects,” and Chapter 13, “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_name1, field_name2 );
```

- ▶ 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 ( string1, string2 );
```

- ▶ 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 and other available **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.

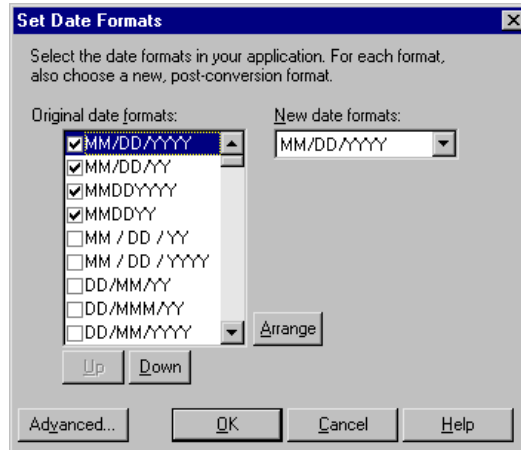
Note: When WinRunner runs tests, it first examines the general settings defined in the Date Operations Run Mode dialog box. Then, it examines the aging overrides for specific date formats. Finally, it considers overrides defined for particular objects.

Overriding Aging of Specific Date Formats

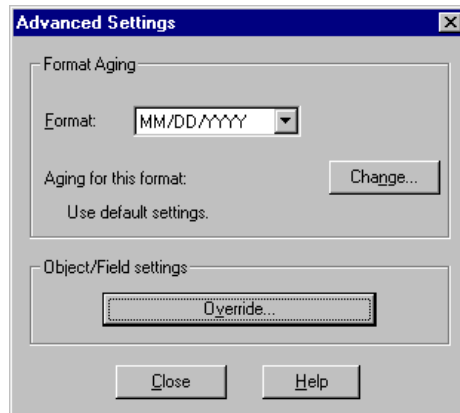
You can override the aging of a specific date format so that it will be aged differently than the default aging setting.

To override the aging of a date format:

- 1 Choose **Tools > Date > Set Date Formats**. The Set Date Formats dialog box opens.



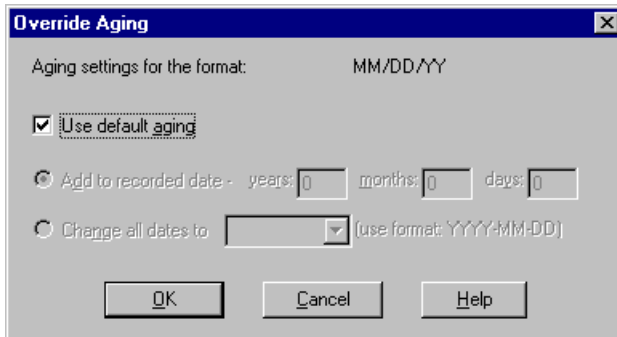
- 2 Click the **Advanced** button. The Advanced Settings dialog box opens.



- 3 In the **Format** list, select a date format.

Note that the **Format** list displays only the date formats that are selected in the **Set Date Formats** dialog box.

- 4 Click **Change**. The Override Aging dialog box opens.



- 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.

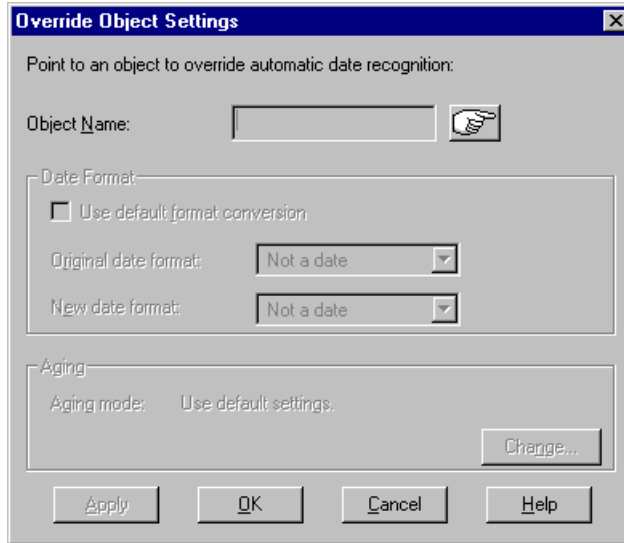
Overriding Aging or Date Format of an Object

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

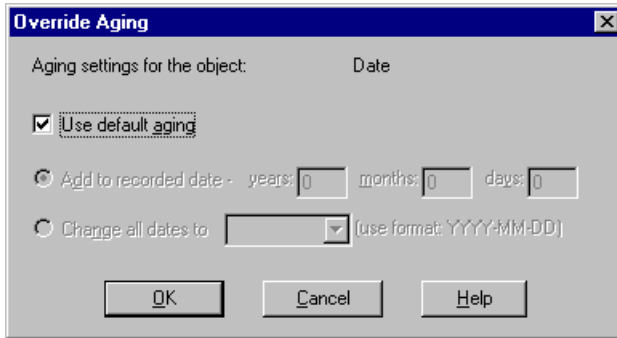
To override settings for an object:

- 1 Choose **Tools > Date > Override Object Settings**. The Override Object Settings dialog box opens.



- 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.



- 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, months, 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*.

18

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
- ▶ The Data-Driven Testing Process
- ▶ Creating a Basic Test for Conversion
- ▶ Converting a Test to a Data-Driven Test
- ▶ Preparing the Data Table
- ▶ Importing Data from a Database
- ▶ Running and Analyzing Data-Driven Tests
- ▶ Assigning the Main Data Table for a Test
- ▶ Using Data-Driven Checkpoints and Bitmap Synchronization Points
- ▶ Using TSL Functions with Data-Driven Tests
- ▶ Guidelines for Creating a Data-Driven Test

About Creating Data-Driven Tests

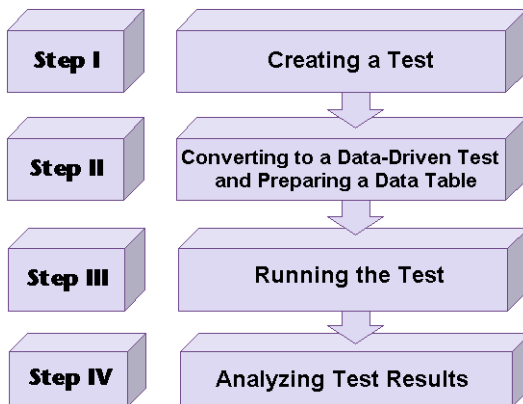
When you test your application, you may want to check how it performs the same operations with multiple sets of data. For example, suppose you want to check how your application responds to ten separate sets of data. You could record ten separate tests, each with its own set of data.

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 variables. The variables 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:

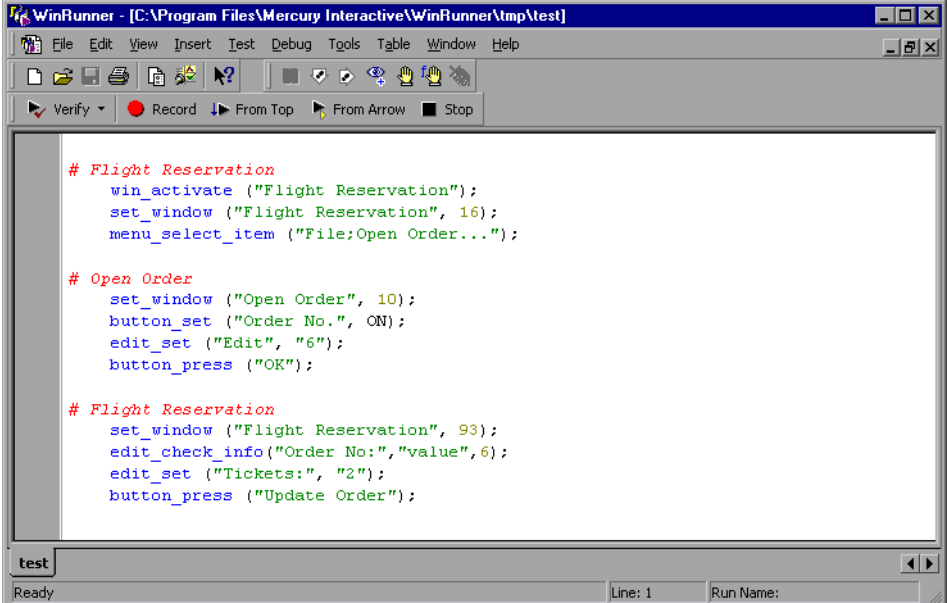


Creating a Basic Test for Conversion

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:



```

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

# 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");

test
Ready Line: 1 Run Name:

```

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 9, “Checking GUI Objects.”

WinRunner can write an event to the Test Results window whenever these statements fail during a test run. To set this option, select the **Fail when single property check fails** check box in the **Run > 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 23, “Setting Global Testing Options,” or refer to Chapter 21, “Setting Testing Options from a Test Script” in the *Mercury WinRunner Advanced Features User’s Guide*.

You can use the **Insert > GUI Checkpoint > For Single Property** command to create property checks using the following *_check_* functions:

button_check_info	scroll_check_info
edit_check_info	static_check_info
list_check_info	win_check_info
obj_check_info	

You can also use the following **_check** functions to check single properties of objects without creating a checklist. You can create statements with these functions manually or using the Function Generator. For additional information, refer to Chapter 8, “Generating Functions” in the *Mercury WinRunner Advanced Features User’s Guide*.

button_check_state	list_check_selected
edit_check_selection	scroll_check_pos
edit_check_text	static_check_text
list_check_item	

For information about specific functions, refer to the *TSL 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 397.

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:

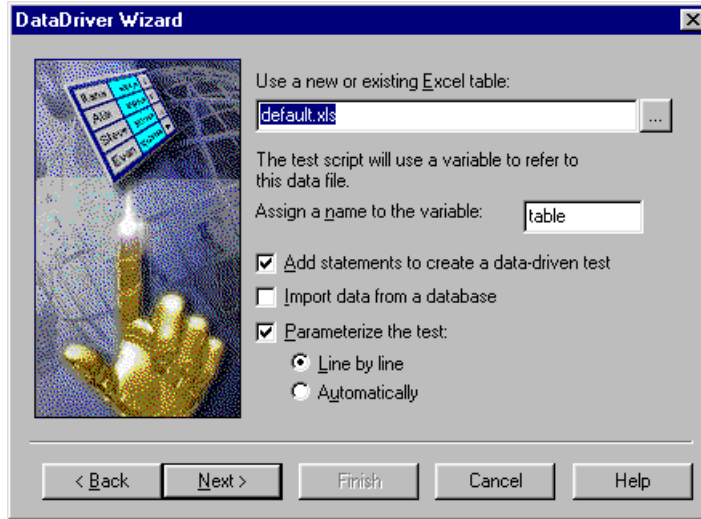
- 1** If you want to turn only part of your test script into a data-driven test, first select those lines in the test script.
- 2** Choose **Table > Data Driver Wizard**.
 - If you selected part of the test script before opening the wizard, proceed to step 3 on page 371.
 - If you did not select any lines of script, the following screen opens:



If you want to turn only part of the test into a data-driven test, click **Cancel**. Select those lines in the test script and reopen the DataDriver wizard.

If you want to turn the entire test into a data-driven test, click **Next**.

3 The following wizard screen opens:



The **Use a new or existing Excel table** box displays the name of the Excel file that WinRunner creates, which stores the data for the data-driven test. 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 379.

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 384.

Note that in order to import data from a database, either Microsoft Query or Data Junction must be installed on your machine. You can install Microsoft Query from the *custom installation* of Microsoft Office. Note that Data Junction is not automatically included in your WinRunner package. To purchase Data Junction, contact your Mercury Interactive representative. For detailed information on working with Data Junction, refer to the documentation in the Data Junction package.

Note: If the **Add statements to create a data-driven test** option is not selected along with the **Import data from a database** option, 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 380.

Note: The `ddt_func.ini` file in the `dat` folder lists the TSL functions that the DataDriver wizard can parameterize while creating a data-driven test. This file also contains the index of the argument that by default can be parameterized for each function. You can modify this list to change the default argument that can be parameterized for a function. You can also modify this list to include user-defined functions or any other TSL functions, so that you can parameterize statements with these functions while creating a test. For information on creating user-defined functions, refer to Chapter 10, “Creating User-Defined Functions” in the *Mercury WinRunner Advanced Features User’s Guide*.

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 384. Otherwise, the following wizard screen opens:



The **Test script line to parameterize** box displays the line of the test script to parameterize. The highlighted value can be replaced by a parameter.

The **Argument to be replaced** box displays the argument (value) that you can replace with a parameter. You can use the arrows to select a different argument to replace.

Choose whether and how to replace the selected data:

- **Do not replace this data:** Does not parameterize this data.
- **An existing column:** If parameters already exist in the data table for this test, select an existing parameter from the list.
- **A new column:** Creates a new column for this parameter in the data table for this test. Adds the selected data to this column of the data table. The default name for the new parameter is the logical name of the object in the selected TSL statement above. Accept this name or assign a new name.

In the sample Flight application test script shown earlier on page 367, 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);
```

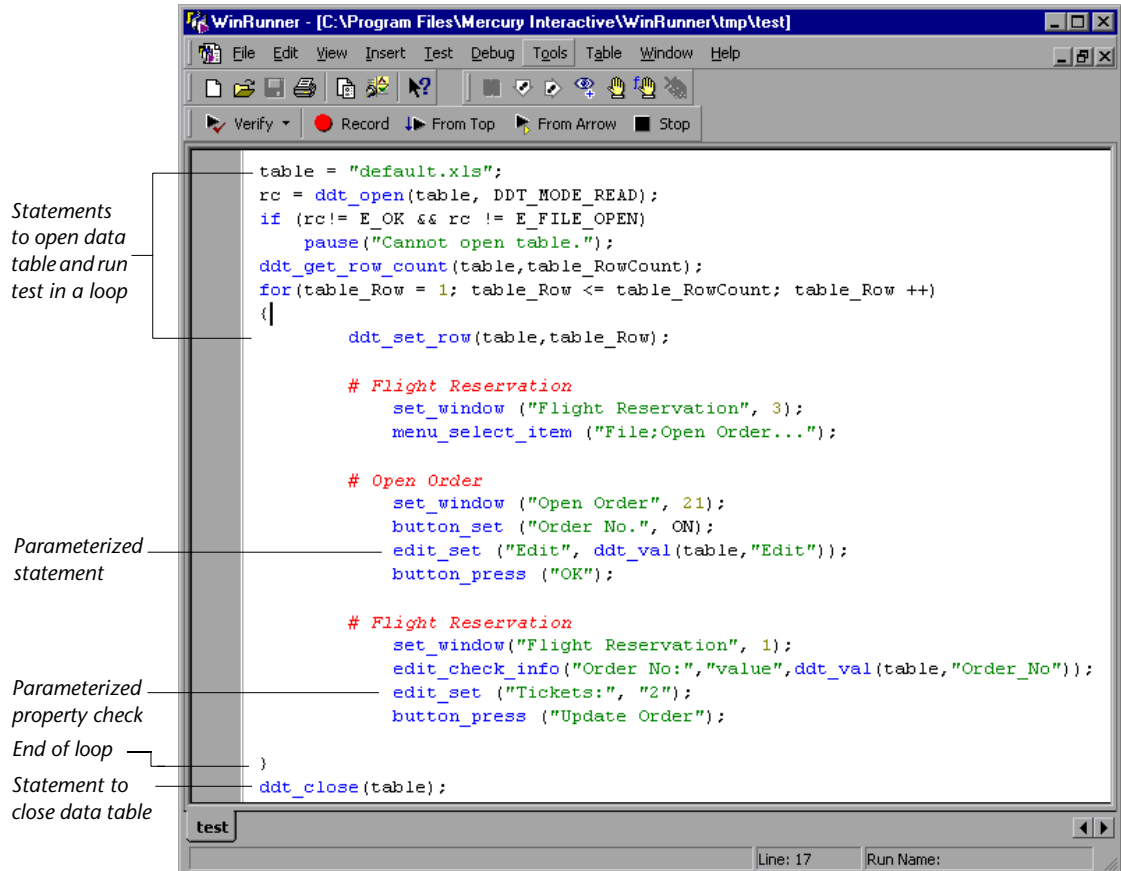
is replaced by:

```
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 367 is modified, as shown below:



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.

The screenshot shows a window titled "Data Table - D:\Program Files\Mercury Interactive\WinR...". The window contains a table with the following structure:

	Edit	Order_No	C	D	E
1	6	6			
2					
3					
4					

The status bar at the bottom of the window displays "Ready".

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 409.

```
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 384. For more information on the `ddt_` functions, see “Using TSL Functions with Data-Driven Tests” on page 402 or refer to the *TSL Reference*.

Parameterizing Values in a Test Script

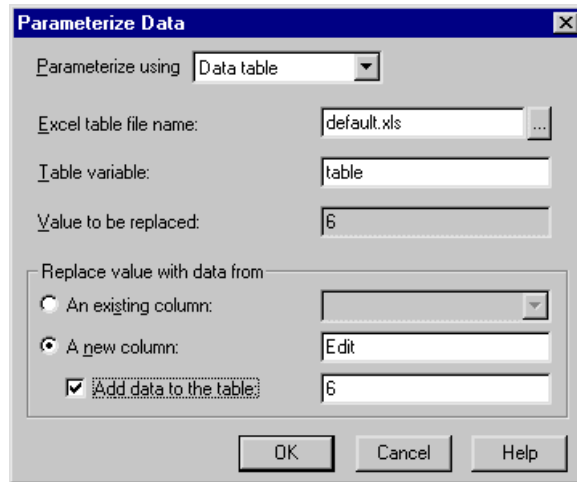
In the sample test script in “Creating a Basic Test for Conversion” on page 367, there are several statements that contain fixed values entered by the user:

```
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**.



- 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 402 or refer to the *TSL Reference*.

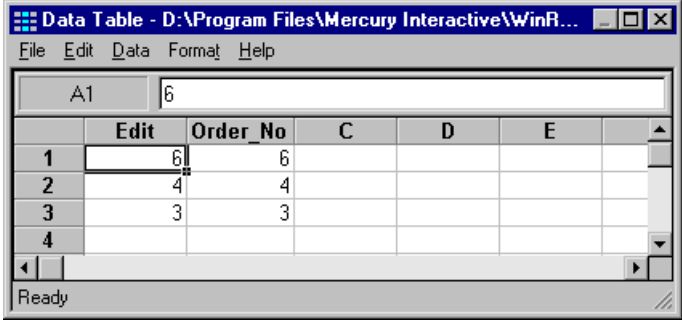
Preparing the Data Table

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.



The screenshot shows a window titled "Data Table - D:\Program Files\Mercury Interactive\WinR...". The window contains a table with the following data:

	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 384. For information on the **ddt_save** function, see “Using TSL Functions with Data-Driven Tests” on page 402. 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 390.

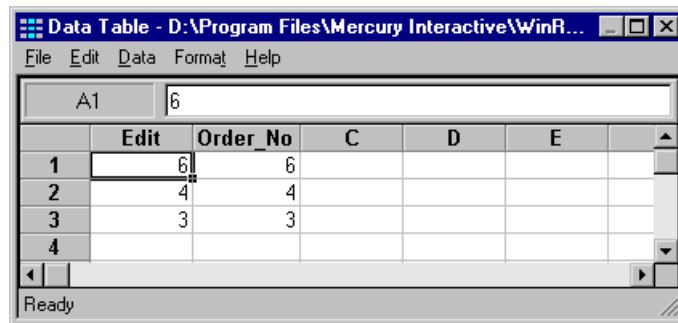
Editing the Data Table

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.



- 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.
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.
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.

Edit Command	Description
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	<p>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.</p> <p>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.</p>

Format Menu

Use the Format menu to set the format of data in a selected cell or cells. The following commands are available in the **Format** menu:

Format Command	Description
General	Sets format to General. General displays numbers with as many decimal places as necessary and no commas.
Currency(0)	Sets format to currency with commas and no decimal places.
Currency(2)	Sets format to currency with commas and two decimal places.
Fixed	Sets format to fixed precision with commas and no decimal places.
Percent	Sets format to percent with no decimal places. Numbers are displayed as percentages with a trailing percent sign (%).
Fraction	Sets format to fraction.
Scientific	Sets format to scientific notation with two decimal places.
Date: (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	1020 characters
maximum row height	409 points
maximum formula length	1024 characters
number precision	15 digits
largest positive number	9.999999999999999E307
largest negative number	-9.999999999999999E307
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.

Setting the Microsoft Query Options

After you select Microsoft Query in the **Connect to database using** option, the following wizard screen opens:



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 392.
- **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 393.
- **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 394.
- **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 374.

Selecting a Source Query File

The following screen opens if you chose **Copy existing query** in the last step.



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 374.

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 374.

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.

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 20, “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 384. For information on the `ddt_update_from_db` function, see “Using TSL Functions with Data-Driven Tests” on page 402 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.

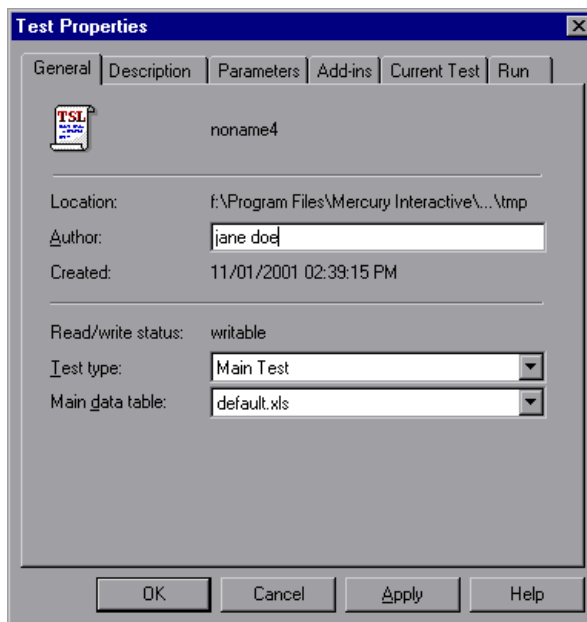
For example, if you inserted a `ddt_report_row` statement in your test script, then WinRunner prints a row of the data table to the test results. Each iteration of a `ddt_report_row` statement in your test script creates a line in the Test Log table in the Test Results window, identified as “Table Row” 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 407 or refer to the *TSL Reference*. For more information on viewing test results, see Chapter 21, “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.



- 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 367. For information on checking a single property of an object, see Chapter 9, “Checking GUI Objects.”

In order to parameterize GUI and bitmap checkpoints and bitmap synchronization points statements, you insert dummy values into the data table for each expected results reference. First you create separate columns for each checkpoint or bitmap synchronization point. Then you enter dummy values in the columns to represent captured expected results. Each dummy value should have a unique name (for example, `gui_exp1`, `gui_exp2`, etc.). When you run the test in Update mode, WinRunner captures expected results during each iteration of the test (i.e. for each row in the data table) and saves all the results in the test's *exp* folder.

- For a GUI checkpoint statement, WinRunner captures the expected values of the object properties.
- For a bitmap checkpoint statement or a bitmap synchronization point statement, WinRunner captures a bitmap.

To create a data-driven checkpoint or bitmap synchronization point:

- 1 Create the initial test by recording or programming.

In the example below, the recorded test opens the Search dialog box in the Notepad application, searches for a text and checks that the appropriate message appears. Note that a GUI checkpoint, a bitmap checkpoint, and a synchronization point are all used in the example.

```
set_window ("Untitled - Notepad", 12);
menu_select_item ("Search;Find...");
set_window ("Find", 5);
edit_set ("Find what:", "John");
button_press ("Find Next");
set_window("Notepad", 10);
obj_check_gui("Message", "list1.ckl", "gui1", 1);
win_check_bitmap("Notepad", "img1", 5, 30, 23, 126, 45);
obj_wait_bitmap("Message", "img2", 13);
set_window ("Notepad", 5);
button_press ("OK");
set_window ("Find", 4);
button_press ("Cancel");
```

- 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 369. Alternatively, you can make these changes to the test script manually. For additional information, see “Creating a Data-Driven Test Manually,” on page 378.

In the example below, the data-driven test searches for several different strings. WinRunner reads all these strings from the data table.

```

set_window ("Untitled - Notepad", 12);
menu_select_item ("Search;Find...");
table = "default.xls";
rc = ddt_open(table, DDT_MODE_READ);
if (rc!= E_OK && rc != E_FILE_OPEN)
    pause("Cannot open table.");
ddt_get_row_count(table,RowCount);
for (i = 1; i <= RowCount; i++) {
    ddt_set_row(table,i);
    set_window ("Find", 5);
    edit_set ("Find what:", ddt_val(table, "Str"));
    button_press ("Find Next");
    set_window("Notepad", 10);

    # The GUI checkpoint statement is not yet parameterized.
    obj_check_gui("message", "list1.ckl", "gui1", 1);

    # The bitmap checkpoint statement is not yet parameterized.
    win_check_bitmap("Notepad", "img1", 5, 30, 23, 126, 45);

    # The synchronization point statement is not yet parameterized.
    obj_wait_bitmap("message", "img2", 13);
    set_window ("Notepad", 5);
    button_press ("OK");
}
ddt_close(table);
set_window ("Find", 4);
button_press ("Cancel");

```

For example, the data table might look like this:

	Str	B	C	D	E
1	John				
2	Susan				
3	Bill				
4					

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:

	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 380. 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, bitmap checkpoints, and bitmap synchronization points in the data table, it recaptures these values from your application and saves 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, refer to Chapter 8, “Generating Functions” in the *Mercury WinRunner Advanced Features User’s Guide*. 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 384.

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 402.
- 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 402.

- 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 397.
- 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.

19

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, refer to Chapter 21, “Setting Testing Options from a Test Script” in the *Mercury WinRunner Advanced Features User’s Guide*. 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 23, “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, refer to Chapter 8, “Generating Functions” in the *Mercury WinRunner Advanced Features User’s Guide*. 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, refer to Chapter 21, "Setting Testing Options from a Test Script" in the *Mercury WinRunner Advanced Features User's Guide*. 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 23, "Setting Global Testing Options."

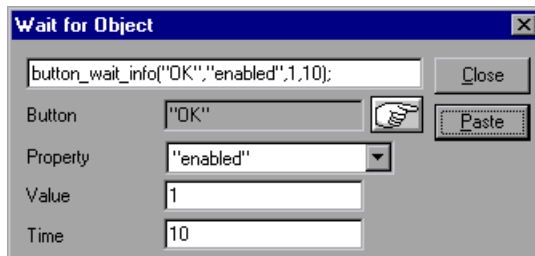
If the window or object you capture has a name that varies from run to run, you can define a regular expression in its physical description in the GUI map. This instructs WinRunner to ignore all or part of the name. For more information, see Chapter 7, “Editing the GUI Map,” and refer to Chapter 6, “Using Regular Expressions” in the *Mercury WinRunner Advanced Features User’s Guide*.

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.



- 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, refer to Chapter 21, “Setting Testing Options from a Test Script” in the *Mercury WinRunner Advanced Features User’s Guide*. 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 23, “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, refer to Chapter 8, “Generating Functions” in the *Mercury WinRunner Advanced Features User’s Guide*. 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, refer to Chapter 21, "Setting Testing Options from a Test Script" in the *Mercury WinRunner Advanced Features User's Guide*.

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 23, "Setting Global Testing Options."

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 the 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, refer to Chapter 21, “Setting Testing Options from a Test Script” in the *Mercury WinRunner Advanced Features User’s Guide*. 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 23, “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, refer to Chapter 21, “Setting Testing Options from a Test Script” in the *Mercury WinRunner Advanced Features User’s Guide*. 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 23, “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, refer to Chapter 21, “Setting Testing Options from a Test Script” in the *Mercury WinRunner Advanced Features User’s Guide*. 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 23, “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 111.
- ▶ **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 397.

Part IV

Running Tests—Basic

20

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
- ▶ WinRunner Test Run Modes
- ▶ WinRunner Run Commands
- ▶ Choosing Run Commands Using Softkeys
- ▶ Running a Test to Check Your Application
- ▶ Running a Test to Debug Your Test Script
- ▶ Running a Test to Update Expected Results
- ▶ Running a Test to Check Date Operations
- ▶ Supplying Values for Input Parameters When Running a Test
- ▶ Controlling the Test Run with Testing Options
- ▶ Solving Common Test Run Problems

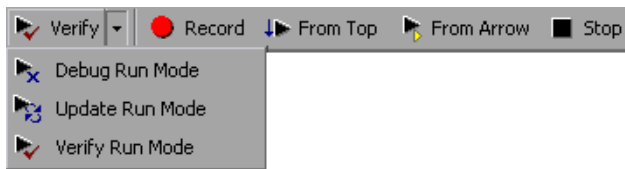
About Understanding Test Runs

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

You choose a run mode from the list on the Test toolbar. The Verify mode is the default run mode for tests.



Note: The Verify run mode is relevant only for tests and is not available when working with components. When working with components, Debug is the default 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, refer to Chapter 14, "Running Batch Tests" in the *Mercury WinRunner Advanced Features User's Guide*.

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 21, “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 436.

Note: Before you 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 431.

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, refer to Chapter 16, “Controlling Your Test Run” in the *Mercury WinRunner Advanced Features User’s Guide*.
- ▶ Set breakpoints at specified points in the test script to pause tests while they run. For more information, refer to Chapter 17, “Using Breakpoints” in the *Mercury WinRunner Advanced Features User’s Guide*.
- ▶ Use the Watch List to monitor variables in a test script while the test runs. For more information, refer to Chapter 18, “Monitoring Variables” in the *Mercury WinRunner Advanced Features User’s Guide*.
- ▶ Use the Call Chain to follow and navigate the test flow. For more information, refer to Chapter 9, “Calling Tests” in the *Mercury WinRunner Advanced Features User’s Guide*.
- ▶ Use the Input Parameters option in the Run dialog box to check how your test handles various parameter values before including the test in a call chain. For more information, refer to Chapter 9, “Calling Tests” in the *Mercury WinRunner Advanced Features User’s Guide*.

For more information about running a test script in Debug mode, see “Running a Test to Debug Your Test Script” on page 437.

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 23, “Setting Global Testing Options,” and refer to Chapter 21, “Setting Testing Options from a Test Script” in the *Mercury WinRunner Advanced Features User’s Guide*.

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 XP and another set of expected results when you run your application in Windows NT. For more information about generating additional sets of expected results, see “Generating Multiple Expected Results” on page 439.

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

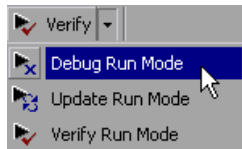
For more information about running a test script in Update mode, see “Running a Test to Update Expected Results” on page 438.

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.

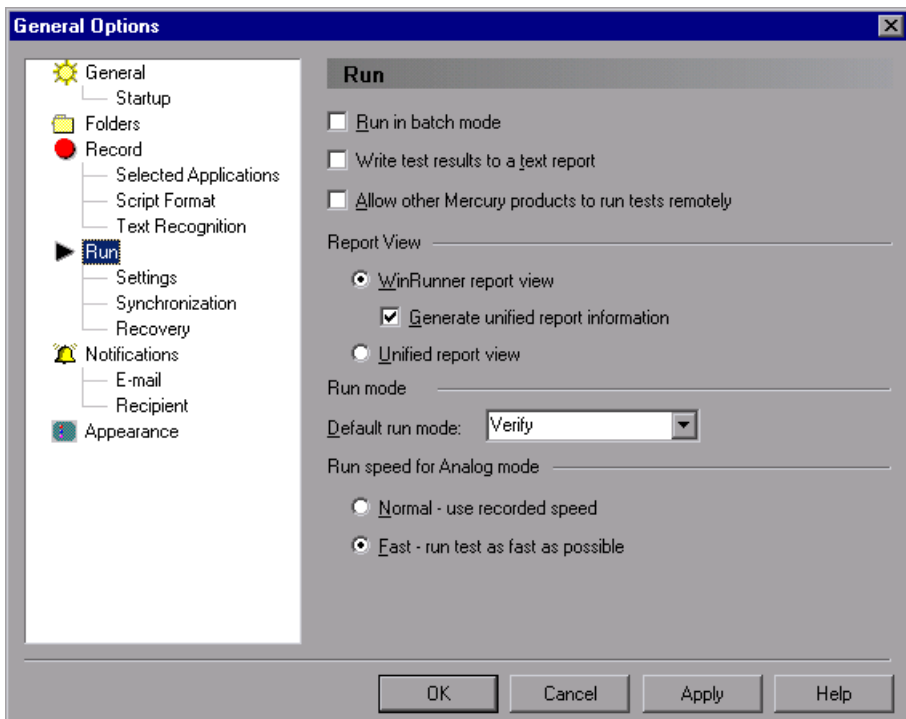
Setting the Default Run Mode

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.



- 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.

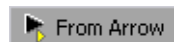
WinRunner Run Commands

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, refer to Chapter 16, “Controlling Your Test Run” in the *Mercury WinRunner Advanced Features User’s Guide*.

The following Step buttons are available:



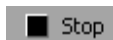
Step button



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 448 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, refer to Chapter 11, “Employing User-Defined Functions in Tests” in the *Mercury WinRunner Advanced Features User’s Guide*.



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 448 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, refer to Chapter 20, “Customizing the WinRunner User Interface” in the *Mercury WinRunner Advanced Features User’s Guide*.

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.

Command	Default Softkey Combination	Function
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

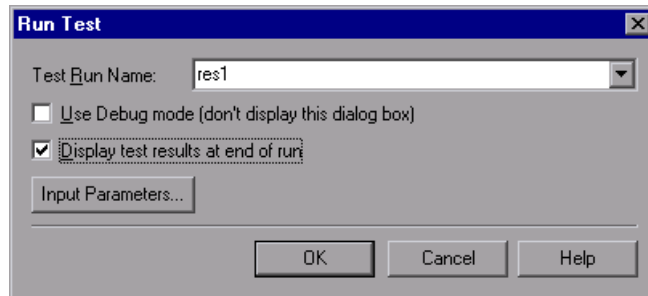
When you run a test to check the behavior of your application, WinRunner compares the current results with the expected results. You specify the folder in which to save the verification results for the test.

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.



- 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.
- 5 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.
- 6 To supply values for input parameters, click the **Input Parameters** button and enter the values you want to use for this test run in the Input Parameters dialog box. For more information, see “Supplying Values for Input Parameters When Running a Test,” on page 447.
- 7 Click **OK**. The Run Test dialog box closes and WinRunner runs the test. 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:

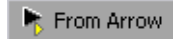


- 1 If your test is not already open, choose **File > Open** to open the test.
- 2 Select **Debug** from the drop-down list of run modes on the Test toolbar.
- 3 Choose the appropriate **Run** or **Debug** menu command.



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:

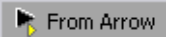


- 1** If your test is not already open, choose **File > Open** to open the test.
- 2** Select **Update** from the list of run modes on the Test toolbar.
- 3** Choose the appropriate **Test** menu command.



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*.

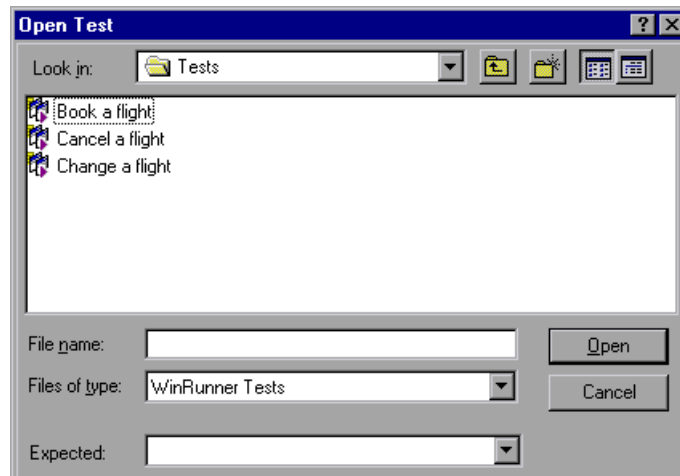
Generating Multiple Expected Results

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:

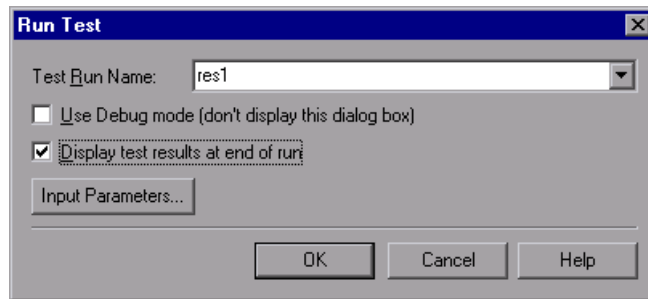


- 1 Choose **File > Open** or click the **Open** button. The Open Test dialog box opens.

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, *res1*.
- 6 To supply values for input parameters, click **Input Parameters** and enter the values you want to use for this test run in the Input Parameters dialog box. For more information, see “Supplying Values for Input Parameters When Running a Test,” on page 447.



- 7 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 17, “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).

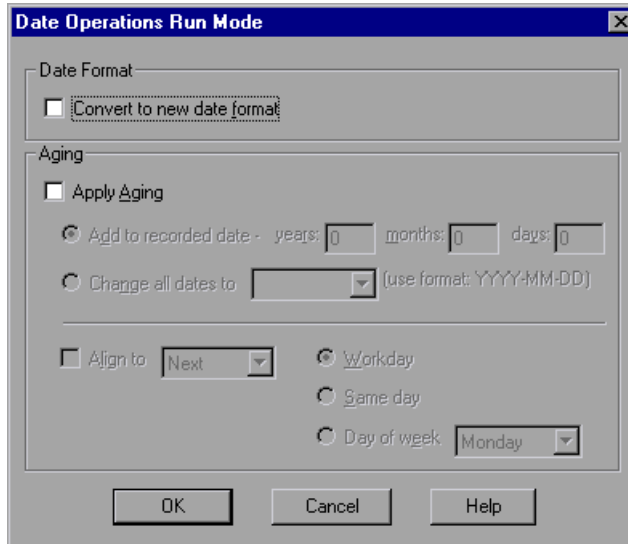
The general run mode settings, Verify, Debug, and Update, are described earlier in this chapter. Note that during a test run in Update mode, dates in the script are not aged or translated to a new format.

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 23, “Setting Global Testing Options.” For more information on the Run Test dialog box, see “Running Tests to Check Date Operations”.

- 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 358.

Running Tests to Check Date Operations

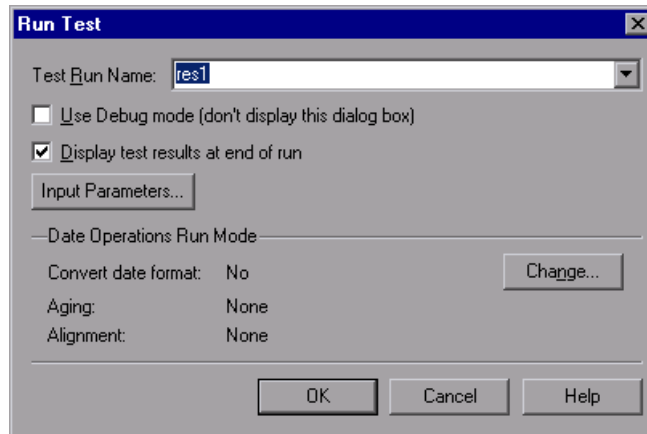
After you set the date operations run mode, you can run your test script.

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 433.

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.



- 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 To supply values for input parameters, click **Input Parameters** and enter the values you want to use for this test run in the Input Parameters dialog box. For more information, see “Supplying Values for Input Parameters When Running a Test” on page 447.
- 6 If you want to change the date operations run mode settings, click **Change** and specify the date operations run mode settings.
- 7 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*.

Supplying Values for Input Parameters When Running a Test

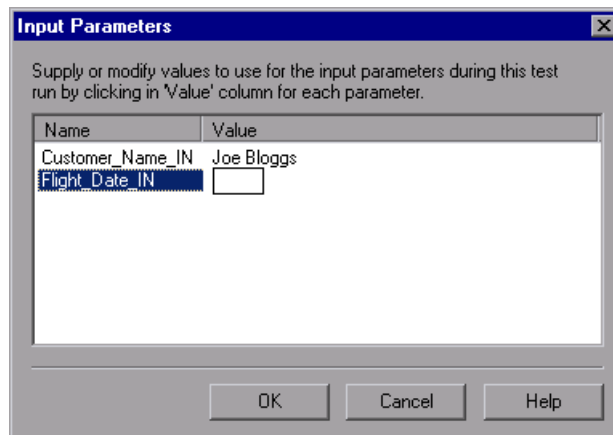
If your test has one or more input parameters defined, you can provide values for those parameters when you start to run your test. These values are used only for the current test run and are not saved with the test.

If you do not supply values for input parameters when you run your test, the default values for the input parameters, if defined, are used. Otherwise, your parameters will receive empty values. The test will run, but steps may fail if they require non-empty values.

For more information on default parameter values, see “Managing Test Parameters” on page 515.

To supply a value for an Input Parameter

- 1 In the Run Test dialog box, click the **Input Parameters** button. The Input Parameters dialog box opens.
- 2 Click the **Value** cell in the row of the Input Parameter whose value you want to supply. Enter the value in the displayed edit area.



- 3 Repeat step 2 for each input parameter whose value you want to supply.
- 4 Click **OK**.

For more information about Input Parameters, refer to Chapter 9, “Calling Tests” in the *Mercury WinRunner Advanced Features User's Guide*.

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 23, “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, refer to Chapter 21, “Setting Testing Options from a Test Script” in the *Mercury WinRunner Advanced Features User's Guide*.

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.



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. Objects stored in a temporary GUI map are not necessarily saved from session to session and you should not rely on their existence in the GUI map after you close 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 <i>load</i> your GUI map file. These steps are described in Chapter 5, “Working in the Global GUI Map File Mode.”

Possible Causes	Possible Solutions
<p>You saved the GUI map file, but it is not loaded.</p>	<p>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.”</p>
<p>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.)</p>	<p>Use a regular expression to enable WinRunner to recognize objects based on a portion of its physical description. For more information on regular expressions, refer to Chapter 6, “Using Regular Expressions” in the <i>Mercury WinRunner Advanced Features User’s Guide</i>.</p>
	<p>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, refer to Chapter 2, “Configuring the GUI Map” in the <i>Mercury WinRunner Advanced Features User’s Guide</i>.</p>
<p>The physical description of the object/window does not match the physical description in the GUI map.</p>	<p>Modify the physical description in the GUI map, as described in “Modifying Logical Names and Physical Descriptions” on page 77.</p>
<p>The logical name of the object/window in the test script does not match the logical name in the GUI map.</p>	<p>Modify the logical name of the object/window in the GUI map, as described in “Modifying Logical Names and Physical Descriptions” on page 77.</p>
	<p>Modify the logical name of the object/window manually in the test script.</p>

Possible Causes	Possible Solutions
<p>The object/window has a different number of obligatory or optional properties (in the GUI map configuration) than in the GUI map.</p>	<p>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 Chapter 2, “Configuring the GUI Map” in the <i>Mercury WinRunner Advanced Features User’s Guide</i>.</p>
	<p>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.”</p>

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 63.

21

Analyzing Test Results

After you run a test or a component from WinRunner, you can view a report of all the major events that occurred during the 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
- ▶ Understanding the Unified Report View Results Window
- ▶ Customizing the Test Results Display
- ▶ Understanding the WinRunner Report View Results Window
- ▶ Viewing the Results of a Test Run
- ▶ Viewing Checkpoint Results
- ▶ Analyzing the Results of a Single-Property Check
- ▶ Analyzing the Results of a GUI Checkpoint
- ▶ Analyzing the Results of a GUI Checkpoint on Table Contents
- ▶ Analyzing the Expected Results of a GUI Checkpoint on Table Contents
- ▶ Analyzing the Results of a Bitmap Checkpoint
- ▶ Analyzing the Results of a Database Checkpoint
- ▶ Analyzing the Expected Results of a Content Check in a Database Checkpoint
- ▶ Updating the Expected Results of a Checkpoint in the WinRunner Report View

- ▶ Viewing the Results of a File Comparison
- ▶ Viewing the Results of a GUI Checkpoint on a Date
- ▶ Reporting Defects Detected During a Test Run

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. The type of results that are displayed depends on the run mode that is currently selected.

Understanding Test Result Views

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 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 Test Result Types

For WinRunner tests, you can view verification, debug, or expected results. For WinRunner components, you can view debug or expected results.

For verification results, only the test name is displayed in the Test Results titlebar. For debug results, **[debug]** is displayed next to the test or component name. For expected results, **[exp]** is displayed next to the test or component name.

When you open the Test Results window in the WinRunner report view, window always displays the results of the most recent run. However, when you open the Test Results window in the Unified report view, the type of results that are displayed correspond to the run mode that is selected in the main WinRunner window when you select to open the results, if such results exist.

For example, if the currently selected test is set to **Verify** run mode, then when you open the Unified report view, the most recent verification results are displayed. If the currently selected component is set to **Debug** run mode, then the debug results are displayed.

If no results exist for your test or component that correspond to the run mode that is currently selected when you open the Unified report view, the Open Test Results dialog box opens over the Test Results window, enabling you to select other results to view.

Note: When working with components, you can view results of an individual component only by opening the Test Results window while the component is open in WinRunner. You cannot browse to results of an individual component using the Open Test Results dialog box. You can use the Open Test Results dialog box to browse for WinRunner test results or for results of a complete business process test.

For more information on run modes, see “WinRunner Test Run Modes” on page 429.

For more information on opening test results, see “Opening Test Results to View a Selected Test Run” on page 477.

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, refer to Chapter 25, “Integrating with QuickTest Professional” in the *Mercury WinRunner Advanced Features User’s Guide*.

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

The screenshot shows the WinRunner Test Results window for a test named 'basic_flight'. The window title is 'basic_flight [res19] - Test Results'. The menu bar includes 'File', 'View', 'Tools', and 'Help'. The toolbar contains various icons for navigation and search. On the left, a 'Results tree' shows a list of test steps: 'start run (basic_flight)', 'property check' (highlighted in blue), '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)', and 'stop run (basic_flight)'. The 'property check' step is marked with a red 'X' icon, indicating failure. The main area displays a 'Results Summary' for 'basic_flight' which is 'Failed'. It provides run details: 'Run started: Wednesday, July 30, 2003 14:58:04', 'Run ended: Wednesday, July 30, 2003 14:58:14', 'Total run time: 00:00:10', and 'Operator name: jackieros'. A table summarizes checkpoint results:

Checkpoint Type	Results	Passed	Failed
Bitmap checkpoints	3	3	0
GUI checkpoints	5	4	1
Database checkpoints	0	0	0

Below the table, an 'Event summary' for the failed 'property check' is shown: 'Event name: property check', 'Result: fail', 'Line number: 19', 'Event time: Wednesday, July 30, 2003 14:58:05', and 'Description: Insert Order:enabled'. A 'Show Event Details' link is provided at the bottom. The status bar at the bottom indicates 'For Help, press F1' and 'Ready'.

For more information on opening the test results window, see “Viewing the Results of a Test Run” on page 474.

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 select options for what to print and how to format the printed results. You can also select a user-defined XSL file with a customized design for the printed report. For more information, see “Printing Test Results” on page 463.

- **Print Preview**—Opens the Print Preview dialog box, which enables you to select options for what and how to display the results information. You can also select a user-defined XSL file with a customized design for the online preview. For more information, see “Previewing Test Results” on page 464.

- **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. For more information, see “Filtering Test Results” on page 462.

- **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 Quality Center and for navigating the test to find steps with a specified result status.



- **Add Defect**—If the Test Results window is already connected to Quality Center, selecting this option opens the Quality Center Add Defect dialog box, which enables you to add a defect to the Quality Center project.

If you are not yet connected, choosing this option opens the Quality Center Connection dialog box. After you connect to Quality Center, the Quality Center Add Defect dialog box opens.



- **Quality Center Connection**—Opens the Quality Center Connection dialog box, which enables you to connect the Test Results window to a Quality Center project.

Note: The unified report viewer is a standalone application. Therefore, even if WinRunner is connected to Quality Center, you must still connect the Test Results window to the Quality Center 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. For more information, see “Finding Results Steps” on page 461.



Once you have set search criteria in the Find dialog box, you can use the **Find Previous** and **Find Next** toolbar buttons to jump to the next or previous step that matches the search criteria.



You can also use the **Go to Previous Node** and **Go to Next Node** toolbar buttons to navigate through your test results report.

- **Help**—Contains options for accessing additional information about the Test Results window.



- **Contents and Index**—Opens the Test Results 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

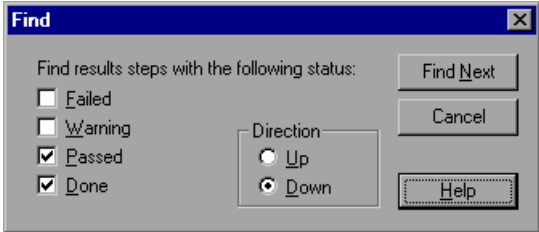
Contains summary information about the event that is currently selected in the results tree, including the event type, status, line number, event time, and a basic description of what was checked.

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**.

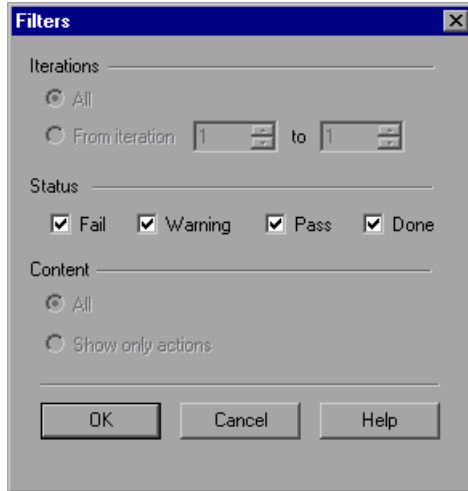


The following options are available:

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.



Note: The **Iterations** and **Content** options are available only from the QuickTest Test Results window. They are not available when viewing test results in WinRunner.

The following options are available:

Option	Description
Status	<ul style="list-style-type: none"> • 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).

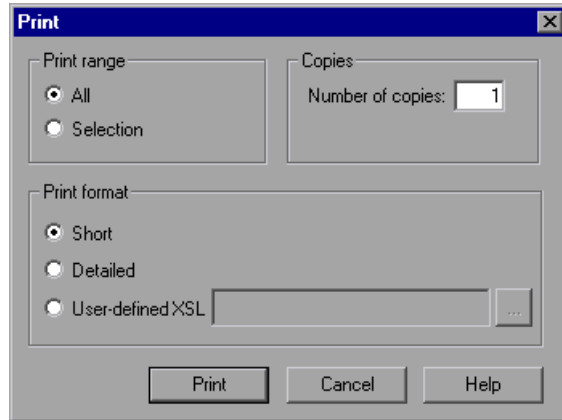
Printing Test Results

You can print test results from the Test Results window. You can select the type of report you want to print, and you can also create and print a customized report.

To print the test results:



- 1 Click the **Print** button or choose **File > Print**. The Print dialog box opens.



- 2 Select a **Print range** option:

- **All**—Prints the results for the entire test or component.
- **Selection**—Prints test results information for the selected branch in the test results tree.

- 3 Specify the **Number of copies** that you want to print.

- 4 Select a **Print format** option:

- **Short**—Prints a summary line (when available) for each item in the test results tree. This option is available only if you selected **All** in step 2.
- **Detailed**—Prints all available information for each item in the test results tree.
- **User-defined XSL**—Enables you to browse to and select a customized **.xsl** file. You can create a customized **.xsl** file that specifies the information to be included in the printed report, and the way it should appear. For more information, see “Customizing the Test Results Display” on page 466.

Note: The **Print format** options are available only for test results created with WinRunner, version 8.0 and later.

- 5 Click **Print** to print the selected test results information to your default Windows printer.

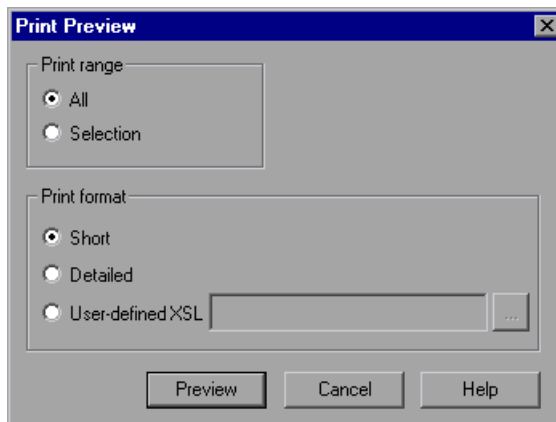
Previewing Test Results

You can preview test results on screen before you print them. You can select the type and quantity of information you want to view, and you can also display the information in a customized format.

Note: The **Print Preview** option is available only for test results created with WinRunner, version 8.0 and later.

To preview the test results:

- 1 Choose **File > Print Preview**. The Print Preview dialog box opens.



2 Select a **Print range** option:

- ▶ **All**—Previews the test results for the entire test or component.
- ▶ **Selection**—Previews test results information for the selected branch in the test results tree.

3 Select a **Print format** option:

- ▶ **Short**—Previews a summary line (when available) for each item in the test results tree. This option is only available if you selected **All** in step 2.
- ▶ **Detailed**—Previews all available information for each item in the test results tree.
- ▶ **User-defined XSL**—Enables you to browse to and select a customized **.xsl** file. You can create a customized **.xsl** file that specifies the information to be included in the preview, and the way it should appear. For more information, see “Customizing the Test Results Display” on page 466.

4 Click **Preview** to preview the appearance of your test results on screen.

Tip: If some of the information is cut off in the preview, for example, if checkpoint names are too long to fit in the display, click the **Page Setup** button in the Print Preview window and change the page orientation from **Portrait** to **Landscape**.

Customizing the Test Results Display

The results of each WinRunner run session are saved in an **.xml** file (called **results.xml**). This **.xml** file stores information about each of the test result nodes in the left pane of the display.

Each node in the test results tree is an element in the **results.xml** file. In addition, there are different elements that represent different types of information displayed in the test results. The sample **results.xml** shows the basic structure of the **results.xml** file. In this image the Step element nodes are collapsed. You can view the child elements and attributes of the Step element by viewing the **results.xml** file of a test containing a variety of different types of steps.

```
- <Report ver="2.0" tmZone="Standard Time">
  <General productName="WinRunner" productVer="8.00" os="Windows 2000" docLocation="D:\
    host="MyComputer" />
- <Doc rID="T0" type="Test">
  - <DName>
    <![CDATA[ MyTest  ]]>
  </DName>
  - <Res>
    <![CDATA[ res1  ]]>
  </Res>
  + <Step rID="T1">
  + <Step rID="T2">
</Doc>
</Report>
```

Note that if your test calls a QuickTest test, then the structure of the nodes under the QuickTest call are somewhat different. For more information on the QuickTest results xml structure, refer to the QuickTest Professional documentation.

You can take test result information from the **.xml** file and use XSL to display the information you require in a customized format, such as for printing or viewing a print preview.

XSL provides you with the tools to describe exactly which test result information to display and exactly where and how to display/print it. You can also modify the **.css** file referenced by the **.xsl** file, to change the appearance of the report (for example, fonts, colors, and so forth).

You may find it easier to modify the existing **.xsl** and **.css** files provided with WinRunner, instead of creating your own customized files from scratch. The files are located in **<WinRunner Installation Folder>\UnifiedReport\dat**, and are named as follows:

- ▶ **PShort.xsl**—Specifies the content of the test results report printed when you select the **Short** option in the Print dialog box.
- ▶ **PDetails.xsl**—Specifies the content of the test results report printed when you select the **Detailed** option in the Print dialog box.
- ▶ **PSelection.xsl**—Specifies the content of the test results report printed when you select the **Selection** option in the Print dialog box.
- ▶ **PResults.css**—Specifies the appearance of the test results print preview. This file is referenced by all three **.xsl** files.

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.



Test name

Menu bar and Toolbar

Results location

Test tree

Test summary

Test log

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

Note: You can customize the background of the Mercury Test Results window. For more information, see “Setting Appearance Options,” on page 585.

For more information on opening the Test Results window, see “Viewing the Results of a Test Run” on page 474.

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 493.



- **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 Quality Center.

► **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 Quality Center project to which you are currently connected. (This option is enabled only when you are connected to a Quality Center 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.



► **Help**—Click the **Help** toolbar button and then click anywhere in the Test Results window to view WinRunner Test Results Help.

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 the *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 checkpoint_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.
checkpoint_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 493.

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)	The 12th 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 484.



► 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 23, “Setting Global Testing Options.”

Viewing the Results of a Test Run

After a test or component run, you can view results in the Test Results window. The Test Results window opens and displays the most recent results of the current test or component. You can view verification (for tests), expected, and debug 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 454 and “Setting Test Run Options” on page 562.
- 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 20, “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 477.
- ▶ 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 Quality Center 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 Quality Center:



- ▶ In the Unified report view—Click the **Quality Center Connection** button or choose **Tools > Quality Center Connection**.
- ▶ In the WinRunner report view—Switch to the WinRunner main window and choose **Tools > Quality Center Connection**.

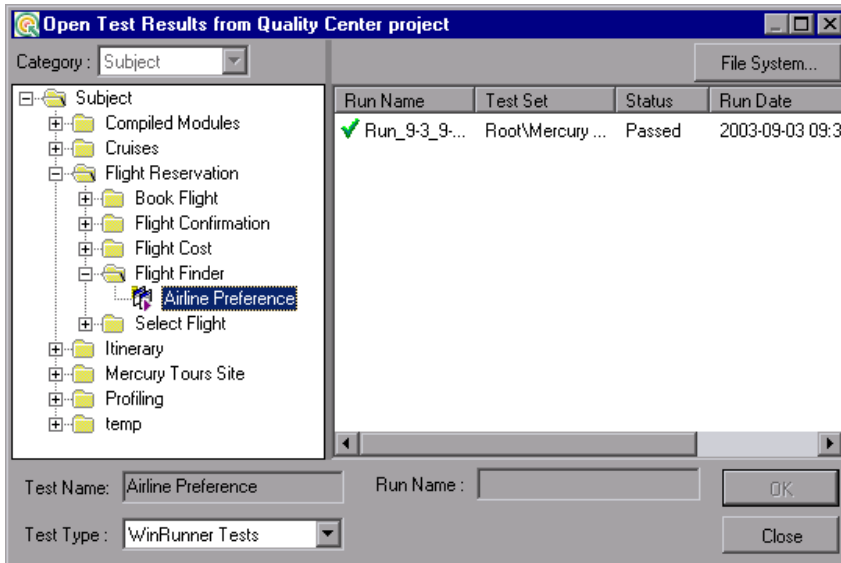
3 Select the Quality Center test results:



- 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 Quality Center Project dialog box opens and displays the test plan tree.



- In the WinRunner report view—Choose **File > Open**. The Open Test Results from Quality Center Project dialog box opens and displays the test plan tree.



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.

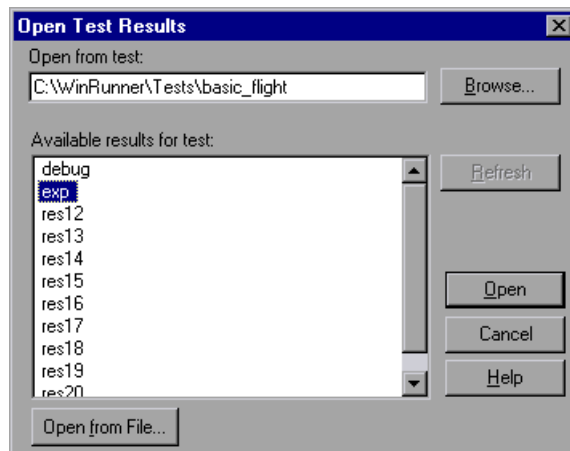
The **Run Date** column displays the date and time when the test set was run.

- 7 Click **OK** to view the results of the selected test.

For more information on viewing the results of a test run from a Quality Center database, refer to Chapter 26, “Managing the Testing Process” in the *Mercury WinRunner Advanced Features User’s Guide*.

Opening Test Results to View a Selected Test Run

You can view the saved results for the current test or component, or you can view the saved results for other WinRunner or business process tests. You select the test results to open for viewing from the Open Test Results dialog box.



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** or **Business process test**.
- 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>\resX\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 Quality Center from the Test Results Window

To manually submit bugs to Quality Center from the Test Results window or to view test results stored in Quality Center, you must be connected to Quality Center.

The connection process has two stages. First, you connect the WinRunner unified report to a local or remote Quality Center Web server. This server handles the connections between WinRunner and the Quality Center project.

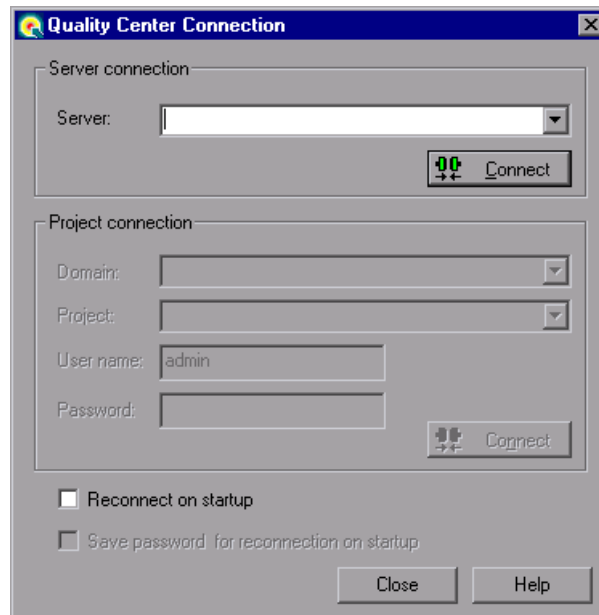
Next, you choose the project in which you want to report the defects.

Note that Quality Center projects are password protected, so you must provide a user name and a password.

To connect the WinRunner unified report to Quality Center:



- 1 Choose **Tools > Quality Center Connection**. The Quality Center Connection dialog box opens.



- 2 In the **Server** box, enter the URL address of the Web server where Quality Center is installed.

Note: You can choose a Web server accessible via a Local Area Network (LAN) or a Wide Area Network (WAN).

- 3 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, or Quality Center, select the domain which contains the TestDirector or Quality Center project in the **Domain** box.

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 Quality Center 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 Quality Center on startup.

- 11 Click **Close** to close the Quality Center Connection dialog box. The Quality Center icon and the address of the Quality Center server are displayed in the status bar to indicate that the WinRunner unified report is currently connected to a Quality Center project.



Tip: You can open the Quality Center Connection dialog box by double-clicking the Quality Center icon in the status bar.

You can disconnect from a Quality Center project and/or server. Note that if you disconnect the WinRunner unified report from a Quality Center 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).

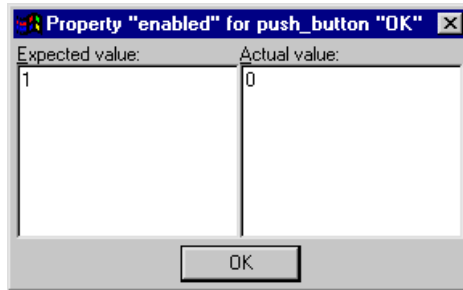
- 3 Double-click the appropriate entry in the test log. Alternatively, highlight the entry and choose **Options > Display** or click the **Display** button. The relevant dialog box opens.
- 4 Click **OK** to close the dialog box.

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.

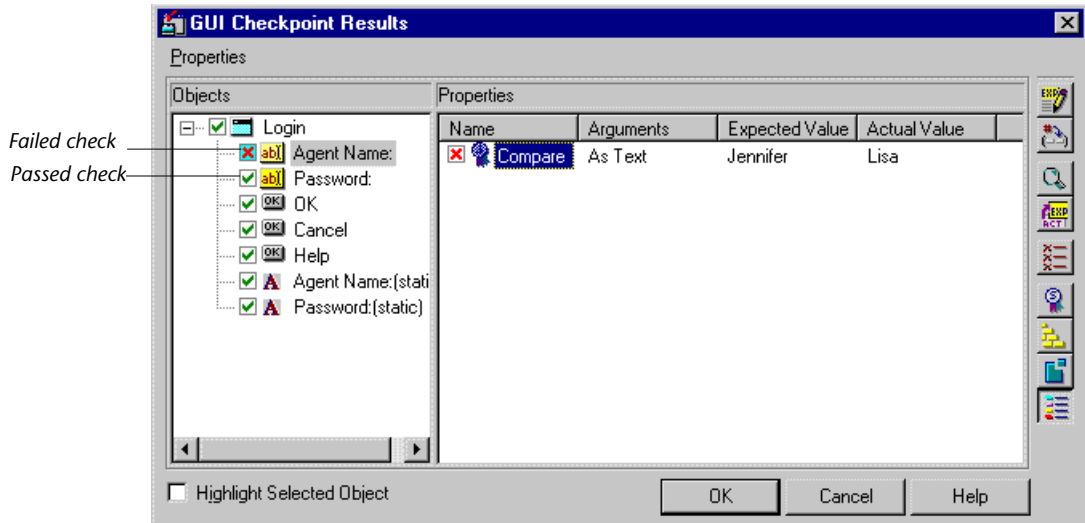
The expected and actual results of a property check are displayed in the Property dialog box that you open from the Test Results window.



For more information, see Chapter 9, "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.












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 499. For a description of other options in this dialog box, see “Options in the GUI Checkpoint Results Dialog Box” on page 485.

For more information, see Chapter 9, “Checking GUI Objects.”

Options in the GUI Checkpoint Results Dialog Box

The GUI Checkpoint Results dialog box includes the following options:

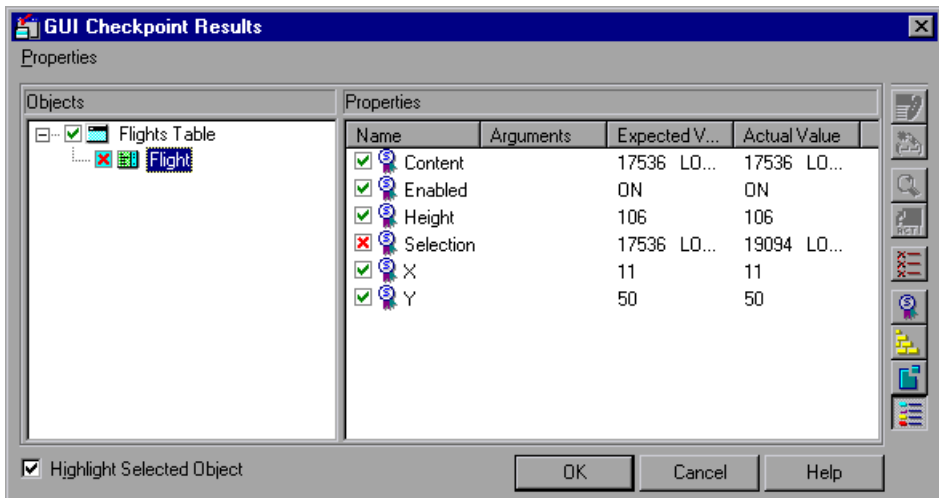
Button	Description
	Edit Expected Value enables you to edit the expected value of the selected property. For more information, see “Editing the Expected Value of a Property” on page 170.
	Specify Arguments enables you to specify the arguments for a check on the selected property. For more information, see “Specifying Arguments for Property Checks” on page 164.
	Compare Expected and Actual Values opens the Compare Values box, which displays the expected and actual values for the selected property check. For a check on table contents, opens the Data Comparison Viewer, which displays the expected and actual values for the check.
	Update Expected Value updates the expected value to the actual value. Note that this overwrites the saved expected value. This option is only available when working in the WinRunner report view.
	Show Failures Only displays only failed checks.
	Show Standard Properties Only displays only standard properties.
	Show Nonstandard Properties Only displays only nonstandard properties, such as Visual Basic, PowerBuilder, and ActiveX control properties.
	Show User Properties Only displays only user-defined property checks. To create user-defined property checks, refer to the <i>WinRunner Customization Guide</i> .
	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 13, “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 481.



- 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.

Cell contains a mismatch.

Cell does not contain a mismatch.

Cell was not included in the comparison.

List of errors and mismatches.

Expected Data				Actual Data					
	Flight	From	Departure	To		Flight	From	Departure	To
1	2457	DEN	10:53 AM	SFO	1	9400	DEN	11:21 AM	LAX
2	9270	DEN	05:21 PM	LAX	2	9270	DEN	05:21 PM	LAX
3	6208	DEN	03:12 PM	LAX	3	6208	DEN	03:12 PM	LAX
4	5439	DEN	12:48 PM	SFO	4	6204	DEN	12:48 PM	LAX
5	6200	DEN	10:24 AM	LAX	5	6200	DEN	10:24 AM	LAX
6	5988	DEN	01:45 PM	LAX	6	5988	DEN	01:45 PM	LAX
7	5595	DEN	04:09 PM	LAX	7	5595	DEN	04:09 PM	LAX
8	5385	DEN	10:09 AM	LAX	8	5385	DEN	10:09 AM	LAX
9	2059	DEN	12:33 PM	LAX	9	2059	DEN	12:33 PM	LAX
10	1513	DEN	06:33 PM	LAX	10	1513	DEN	06:33 PM	LAX
11	1159	DEN	02:57 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'.

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 253.



- 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”).

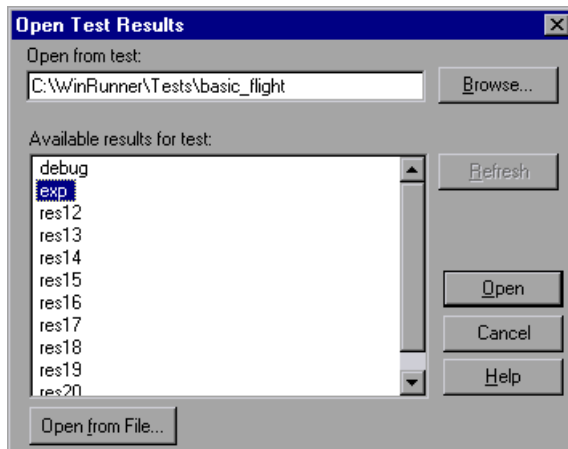
Note that you can also view the expected results of a GUI checkpoint on a table from the Edit Check dialog box. For additional information, see Chapter 13, “Checking Table Contents.”

To display the expected results of a GUI checkpoint on table contents:

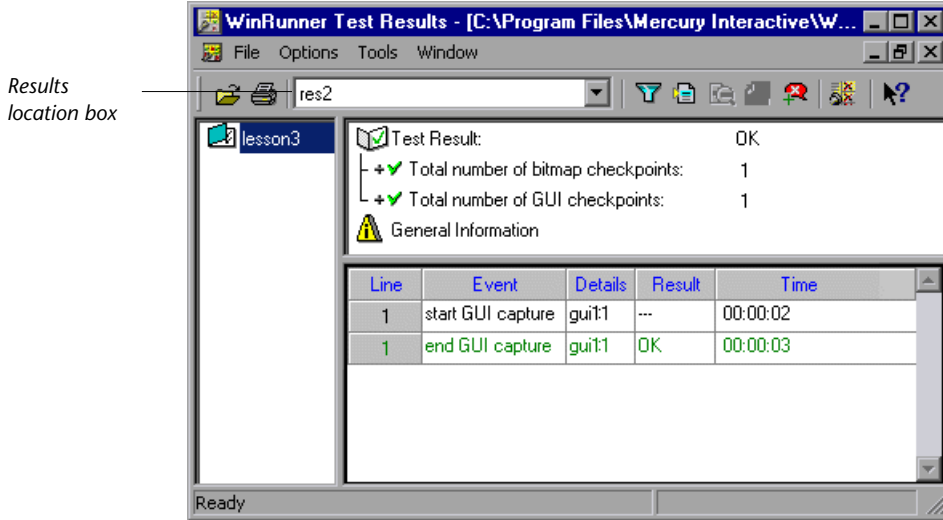
- 1 Open the Test Results window and display the test for which you want to view expected results. For more information, see “Viewing Checkpoint Results” on page 481.
- 2 Display the expected results:



- 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.



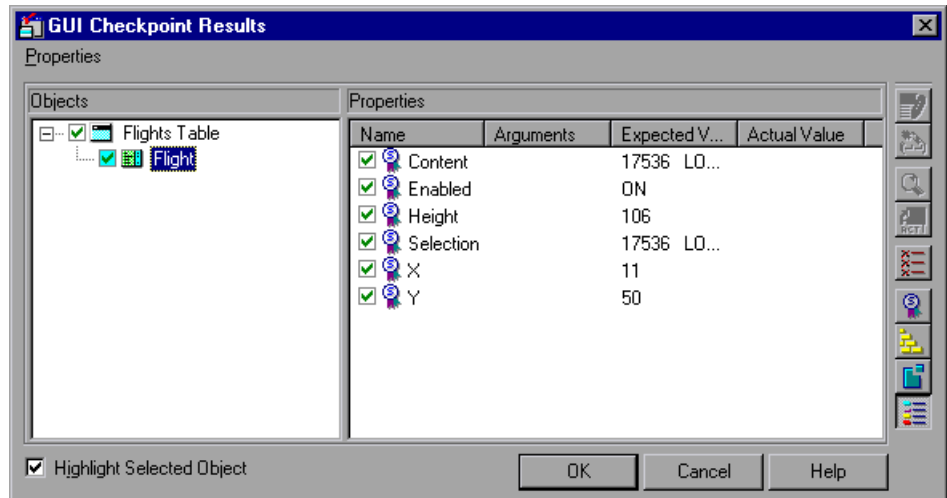
3 Display the expected results:

- 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.

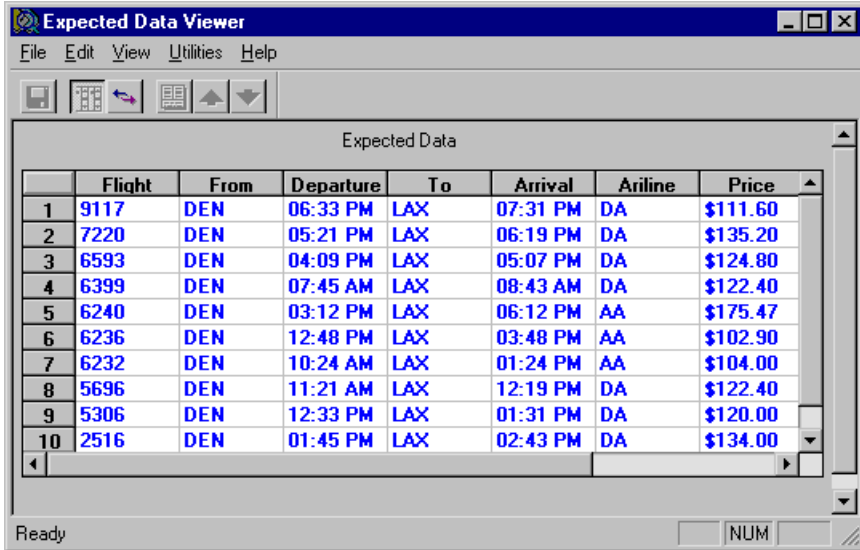


Note: Since you are viewing the *expected* results of the GUI checkpoint, the *actual* values are not displayed.



- 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.



The screenshot shows a window titled "Expected Data Viewer" with a menu bar (File, Edit, View, Utilities, Help) and a toolbar. The main area displays a table of flight data with the following columns: Flight, From, Departure, To, Arrival, Airline, and Price. The data is as follows:

	Flight	From	Departure	To	Arrival	Airline	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

The status bar at the bottom shows "Ready" and a "NUM" field.

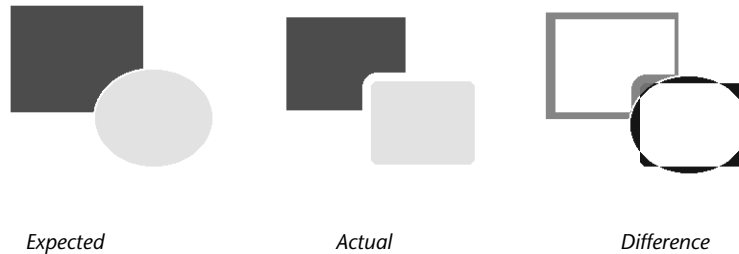
Note: When working in the WinRunner report view, you can edit the data in the Edit Check dialog box, which you open from the GUI Checkpoint Results dialog box. To do so, highlight the **TableContent** (or corresponding) property check, and click the **Edit Expected Value** button. For information on working with the Edit Check dialog box, see “Understanding the Edit Check Dialog Box” on page 253.



- 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.

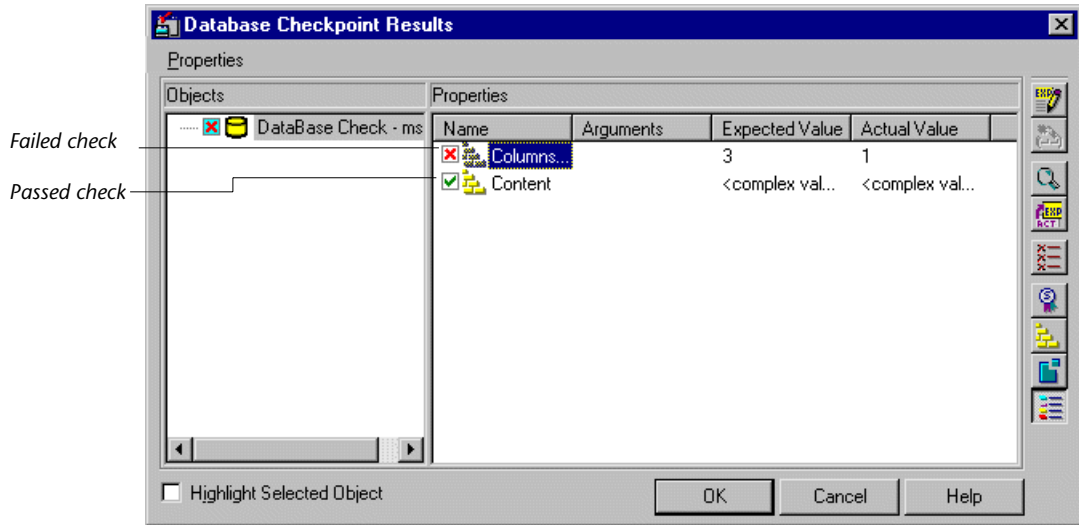


When viewing results in the WinRunner report view, you can control which types of bitmaps are displayed (expected, actual, difference) when you view the results of a bitmap checkpoint. To set the controls, choose **Options > Bitmap Controls** in the Test Results window.

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 324.

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.



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, see “Updating the Expected Results of a Checkpoint in the WinRunner Report View” on page 499. For a description of other options in this dialog box, see “Options in the Database Checkpoint Results Dialog Box” on page 495.










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 291.

For more information, see Chapter 14, “Checking Databases.”

Options in the Database Checkpoint Results Dialog Box

The Database Checkpoint Results dialog box includes the following options:

Button	Description
	Edit Expected Value enables you to edit the expected value of the selected property. For more information, see “Creating a Custom Check on a Database” on page 280.
	Compare Expected and Actual Values opens the Compare Values box, which displays the expected and actual values for the selected property check. For a Content check, opens the Data Comparison Viewer, which displays the expected and actual values for the check.
	Update Expected Value updates the expected value to the actual value. Note that this overwrites the saved expected value. This option is only available when working in the WinRunner report view.
	Show Failures Only displays only failed checks.
	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 content checkpoint from the Edit Check dialog box. For additional information, see Chapter 14, “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 481.

- 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.

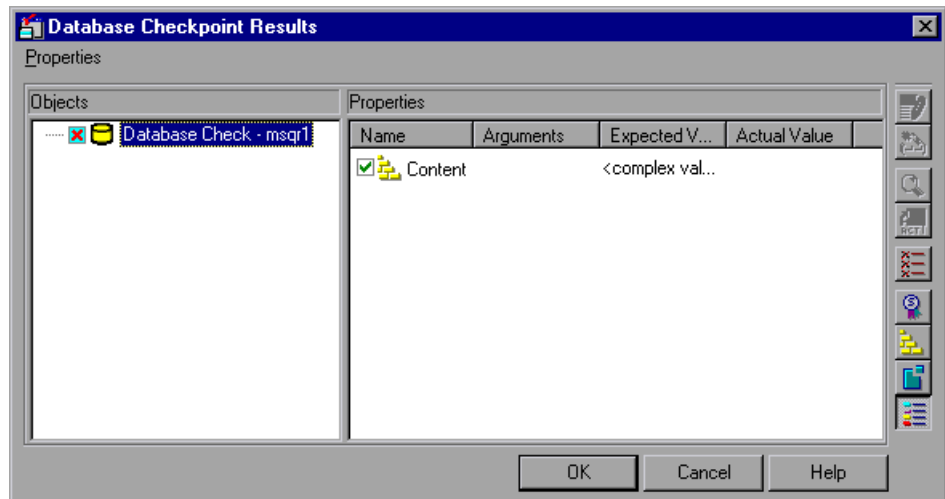
Note that since you are viewing the *expected* results of a test, the total number of database checkpoints performed is listed as zero.

3 Display the expected results:

- In the Unified report view—Click the results tree node for the database 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.

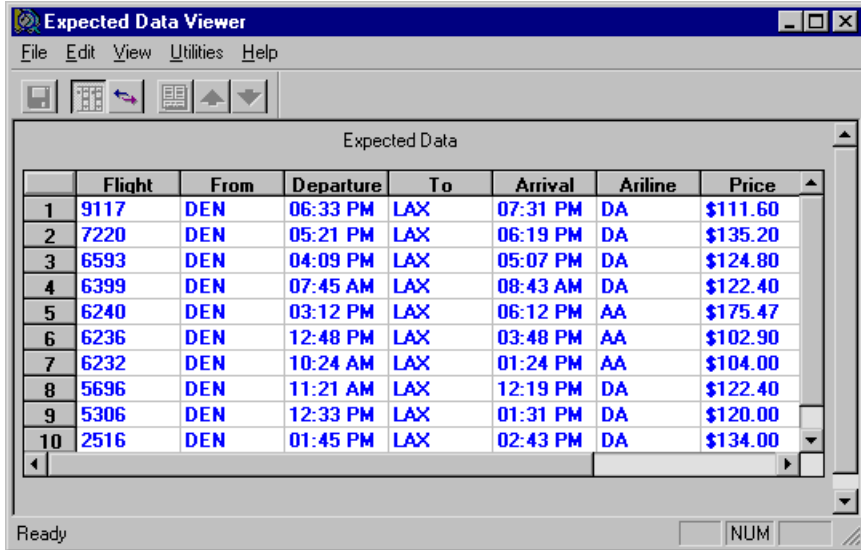


Note that since you are viewing the *expected* results of the database checkpoint, the *actual* values are not displayed.



- ### 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.



The screenshot shows a window titled "Expected Data Viewer" with a menu bar (File, Edit, View, Utilities, Help) and a toolbar. The main area displays a table of flight data under the heading "Expected Data". The table has columns for Flight, From, Departure, To, Arrival, Airline, and Price. The data is as follows:

	Flight	From	Departure	To	Arrival	Airline	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

The status bar at the bottom shows "Ready" and a "NUM" field.

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 291.




- 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 481.
- 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**.

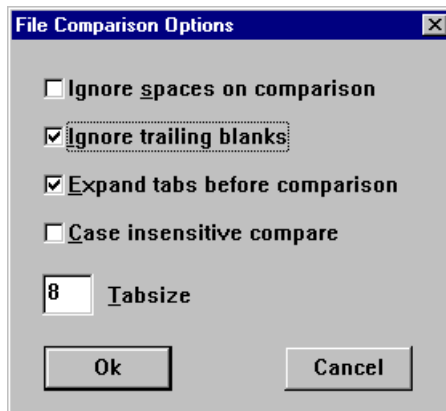


The WDiff utility window opens.

File	Script 1	Script 2
C:\mercury\wrun40\tmp\Sample 1\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.", 0N);	button set ("Order No.", 0N);
	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.MaskedTextBox", "111199");	obj_type ("MSMask.MaskedTextBox", "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.



Note that when you modify any of the options, the two files are read and compared again.

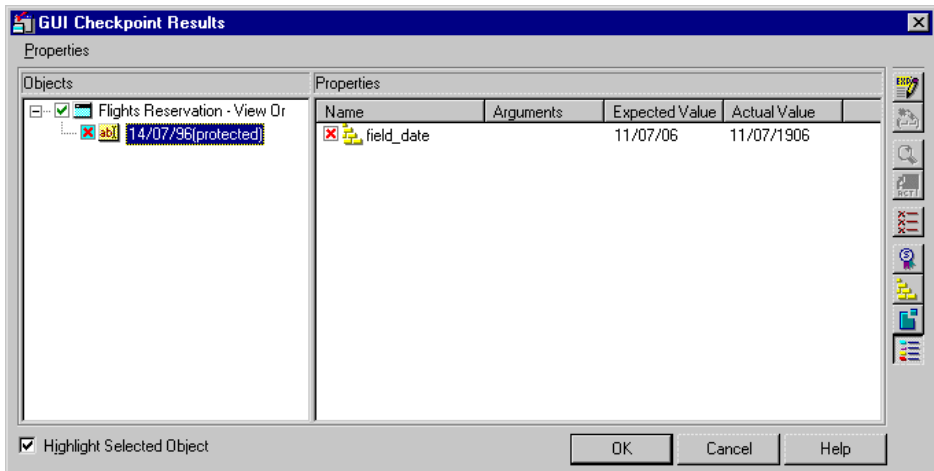
- **Ignore spaces on comparison:** Tab characters and spaces are ignored on comparison.
- **Ignore trailing blanks (default):** One or more blanks at the end of a line are ignored during the comparison.

- ▶ **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.

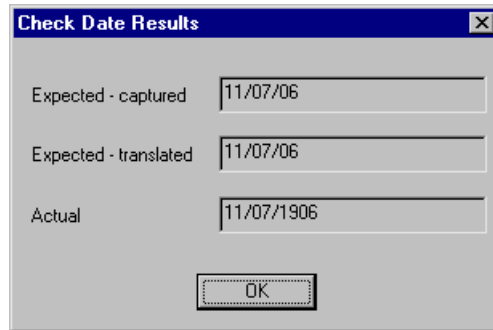
Viewing the Results of a GUI Checkpoint on a Date

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.





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.



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 Quality Center 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 Quality Center 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. (The Add Defect option is supported only when working with TestDirector 7.2 or Quality Center.)

Using the Add Defect Dialog Box

The Add Defect dialog box is a defect tracking component of Quality Center, Mercury Interactive's Web-based test management tool. You can report application defects directly to a Quality Center 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 Test Director 7.2 or Quality Center is installed and that WinRunner is connected to a Quality Center 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 Quality Center 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 Quality Center, see "Connecting to Quality Center from the Test Results Window" on page 479.

For more information about installing Quality Center, refer to the *Mercury Quality Center Installation Guide*.

Reporting Defects with the Add Defect Dialog Box

When you are connected to Quality Center, 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 Quality Center from the main WinRunner window. For more information, see "Connecting to Quality Center from the Test Results Window" on page 479.

If you are working in the Unified report view, you can connect to Quality Center directly 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 481.
- 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 Quality Center, the Quality Center Connection dialog box opens. Connect to Quality Center as described in “Connecting to Quality Center from the Test Results Window” on page 479. When you are finished, click **Close** to close the Quality Center 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 Quality Center 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. Quality Center assigns the new defect a Defect ID.
- 11 Click **Close**.

For more information on using the Add Defect dialog box, refer to the *Mercury Quality Center User's Guide*.

Reporting Defects During a Test Run

You can insert `qcdb_add_defect` statements to your test to instruct WinRunner to add a defect to a Quality Center 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 `qcdb_add_defect` statements:

```
qcdb_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 the Quality Center documentation.

If your test contains `qcdb_add_defect` statements, confirm that you are connected to the appropriate Quality Center project before running your test.

Part V

Configuring Basic Settings

22

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 that 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.

You can also set testing options that affect all tests. For more information, see Chapter 23, “Setting Global Testing Options.”

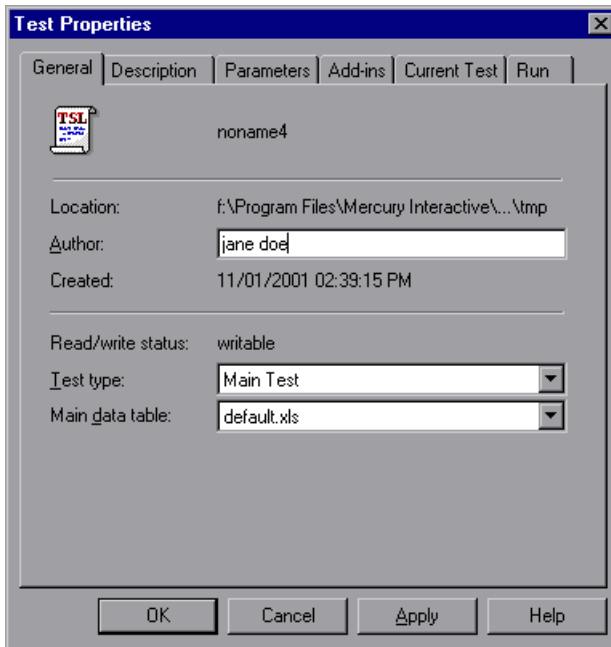
Setting Test Properties from the Test Properties Dialog Box

You can set test-specific properties for any open test in the Test Properties dialog box.

To set test properties:

- 1 Choose **File > Test Properties**.

The Test Properties dialog box opens. It is divided by subject into six tabbed pages.



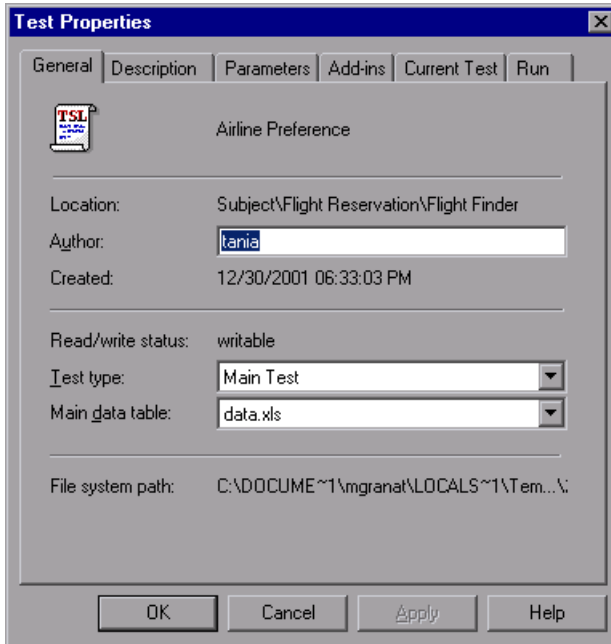
- 2 To set the properties for your test, select the appropriate tab and set the options, as described in the sections that follow.
- 3 To apply your changes and keep the Test Properties dialog box open, click **Apply**.
- 4 When you are finished, click **OK** to save your changes and close the dialog 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 input and output 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.

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. You can also specify a Microsoft Excel file to use for the test's input data and you can view other summary information.



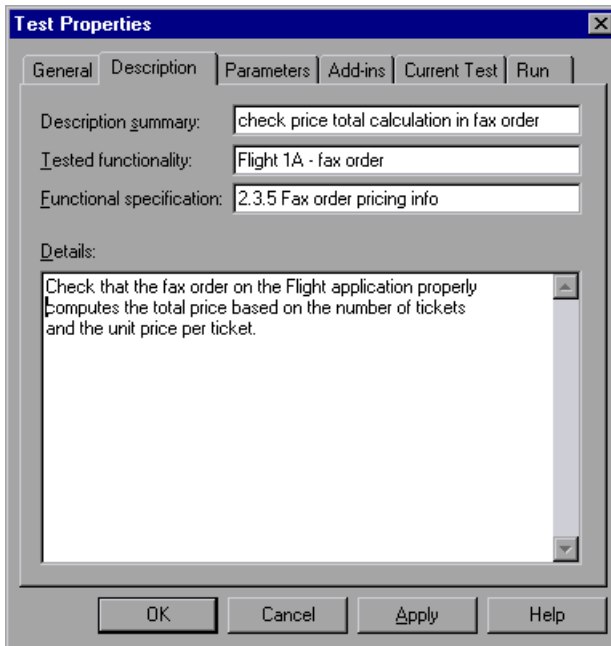
This tab contains the following information:

Option	Description
	Displays the name of the test.
Location	Displays the test's location within the file system or in the Quality Center tree.
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 Test (standard test) or a Compiled Module . For more information about compiled modules, refer to Chapter 11, “Creating a Compiled Module” in the <i>Mercury WinRunner Advanced Features User's Guide</i> .
Main data table	Indicates the main data table for the test. For more information, see “Assigning the Main Data Table for a Test,” on page 396.
File system path	Displays the system file path of the test. This information is displayed only when you are connected to Quality Center and the current test is opened from a Quality Center project.
Version control	Displays version control information for the test. This information is displayed only when you are connected to Quality Center project that supports version control.

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, a reference to the relevant functional specifications document(s), and additional details about the purpose, contents, or requirements of the test.



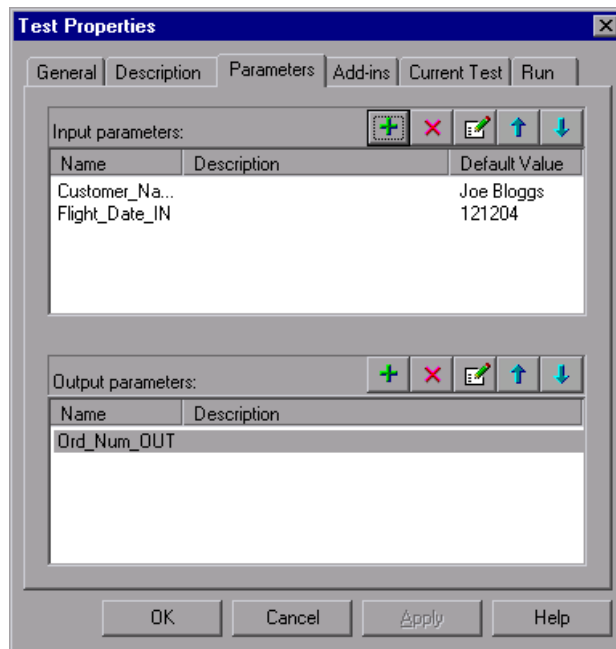
This tab contains the following information:

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.

Option	Description
Functional specification	Enables you to specify a reference to the application's functional specification(s) for the features you are testing.
Details	Enables you to enter 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 input parameters that are listed in the **Parameters** tab are assigned the values supplied by the calling test, and the output parameters return values, generated within the current test, to the calling test.

You can assign default values for input parameters. The default value for an input parameter is used if the calling test does not pass a value for the input parameter in its test call.

You must declare your test parameters in this dialog box to receive input parameter values from a calling test, and return output parameters to a calling test. The order in which parameters are listed in this tab determines the order in which a calling test must supply the parameters. In the test call, input parameters come before output parameters.

Tip: If you add, delete, or modify the order of parameters for a test that is already called by other WinRunner tests or by other Mercury products, ensure that you adjust the parameters accordingly in the calling test or product.

Note: Test parameters are used only in tests of type Main Test. They are not used in compiled modules.

For more information about parameters, refer to Chapter 9, “Working with Test Parameters” in the *Mercury WinRunner Advanced Features User’s Guide*.



To define a new input or output parameter:

- 1 In the **Parameters** tab of the Test Properties dialog box, click the **Add** button corresponding to the parameter list (**Input** or **Output**) to which you want to add a parameter.

The Input Parameters or Output Parameters dialog box opens. For input parameters, the dialog box includes a text box to enter a **Default Value**.

For output parameters, there is no **Default Value** edit box in the dialog box.

- 2 Enter a **Name** and a **Description** for the parameter. For input parameters, you can specify a **Default Value** for the parameter.

Tip: It is recommended to use IN or OUT prefixes or suffixes for the parameter names to indicate the parameter type. This makes your test more readable and makes it easier for other test designers to determine what to enter in call statements to your test.

- 3 Click **OK**. The parameter is added to the appropriate **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. In test calls, input parameters always come before output parameters.

- 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 corresponding to the parameter type you want to delete.
- 3 Click **OK** to close the dialog box.



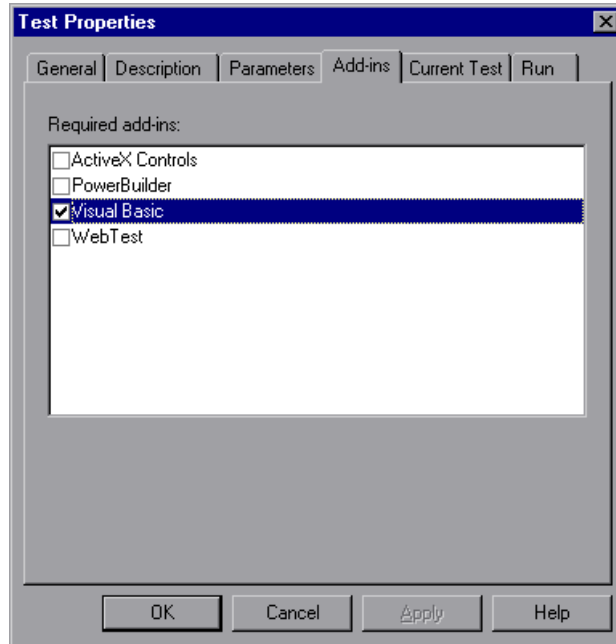
To modify a parameter in the parameter list:

- 1 In the **Parameters** tab of the Test Properties dialog box, select the name of the parameter to modify.
- 2 Click the **Modify Parameter** button or double-click the parameter name. 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.



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 should be loaded to successfully run this test. It also instructs Quality Center to confirm that the required add-ins are loaded. For more information, see “Running Tests with Add-ins from Quality Center” on page 520.

Note: You can see which add-ins are loaded at any time in the **About WinRunner** dialog box (**Help > About**). 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 Quality Center

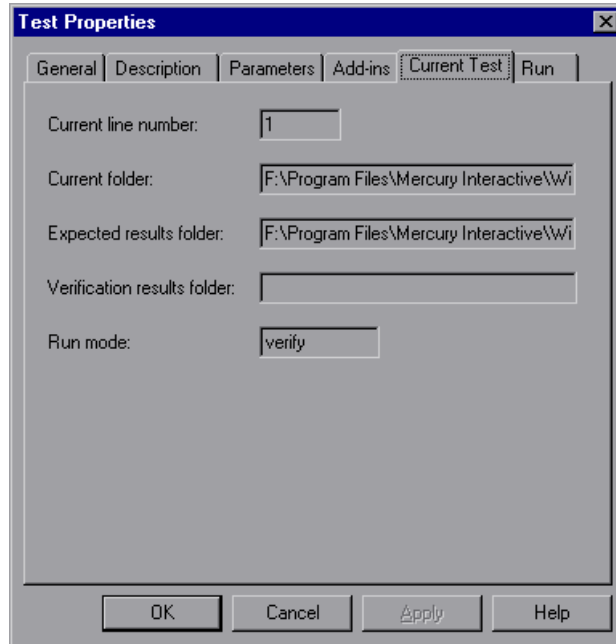
In addition to providing information for people running your test from WinRunner, the **Add-ins** tab instructs Quality Center to load the selected add-ins when it runs WinRunner tests.

When you run a test from Quality Center, Quality Center loads 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, Quality Center closes and reopens WinRunner with the proper add-ins. If one or more of the required add-ins are not installed, Quality Center displays the error message, **Cannot open test**.

For more information about running WinRunner tests from Quality Center, refer to the *Mercury Quality Center 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

This box displays the line number corresponding to the current location of the execution arrow in the test script.

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 21, “Setting Testing Options from a Test Script” in the *Mercury WinRunner Advanced Features User’s Guide*.

Current folder

This box displays the current working folder for the test.

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 21, “Setting Testing Options from a Test Script” in the *Mercury WinRunner Advanced Features User’s Guide*.

Expected results folder

This box displays the full path of the expected results folder associated with the current test run.

You can use the **getvar** function to retrieve the value of the corresponding *exp* testing option from within a test script, as described in Chapter 21, “Setting Testing Options from a Test Script” in the *Mercury WinRunner Advanced Features User’s Guide*.

You can also set this option using the corresponding **-exp** command line option, described in Chapter 15, “Running Tests from the Command Line” in the *Mercury WinRunner Advanced Features User’s Guide*.

Verification results folder

This box displays the full path of the verification results folder associated with the current test run.

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 21, “Setting Testing Options from a Test Script” in the *Mercury WinRunner Advanced Features User’s Guide*.

Run mode

This box displays the current run mode: Verify, Debug, or Update.

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 21, “Setting Testing Options from a Test Script” in the *Mercury WinRunner Advanced Features User’s Guide*.

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**. For more information on these options, see “WinRunner Run Commands” on page 433.

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, refer to Chapter 9, “Calling Tests” in the *Mercury WinRunner Advanced Features User’s Guide*. For more information on the **Step Into** option, refer to Chapter 16, “Controlling Your Test Run” in the *Mercury WinRunner Advanced Features User’s Guide*.

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.

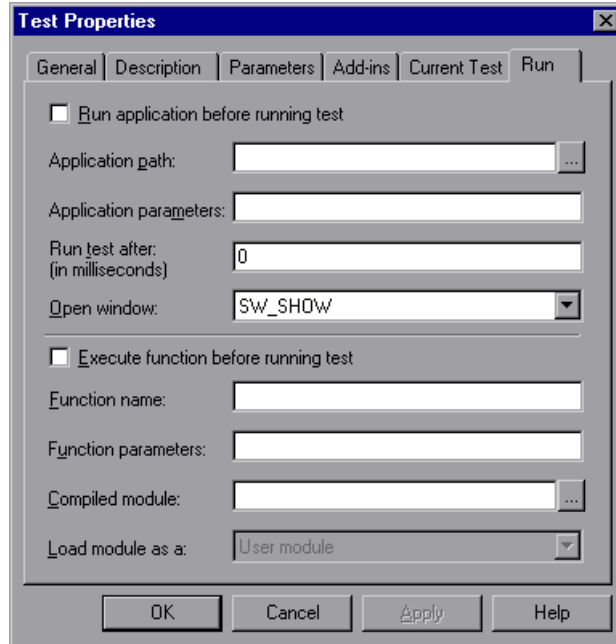
Note that additional methods exist for running an application:

- 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, refer to Chapter 7, “Enhancing Your Test Scripts with Programming” in the *Mercury WinRunner Advanced Features User’s Guide*.
- 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, refer to Chapter 15, “Running Tests from the Command Line” in the *Mercury WinRunner Advanced Features User’s Guide*.

Note: If the application specified as the startup application is already running when you run your test, WinRunner does 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.



- 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. Enter the full path of the application. Do not use quotation marks.

You can specify only **.exe** and **.com** files. If you want to run a file with another extension, specify the **.exe** or **.com** application that will contain the file in the **Application path** box. Then specify the file name in the **Application parameters** box.

For example, suppose you want to run an **.htm** file. Type in the path for your browser in the **Application path** box – e.g. `C:\Program Files\Internet Explorer\IEXPLORE.EXE`. Then type in the full path of the **.htm** file in the **Application parameters** box.

- 5 Enter any required application parameters in the **Application parameters** box, separated by commas (,). The text in the **Application parameters** box may be in quotation marks. 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 open. 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	Minimizes 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.

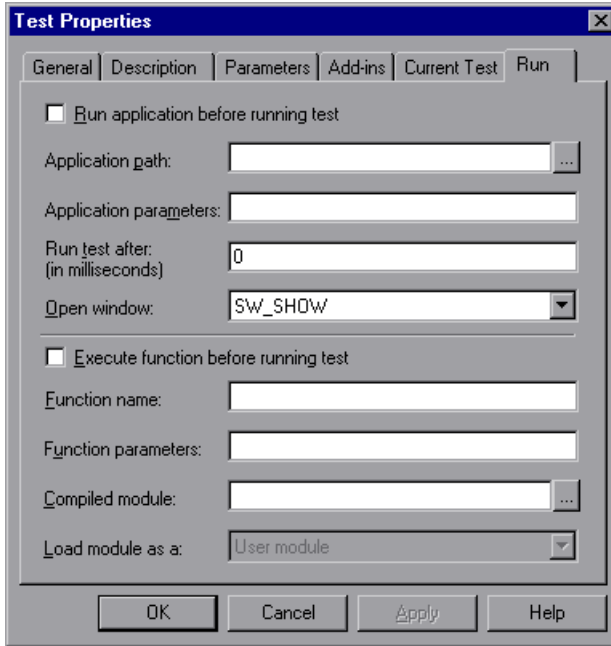
Note: You can also set this option using the `-app_open_win` command line option. For more information, refer to Chapter 15, “Running Tests from the Command Line” in the *Mercury WinRunner Advanced Features User’s Guide*.

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, refer to Chapter 8, “Generating Functions” in the *Mercury WinRunner Advanced Features User’s Guide* and the *TSL Reference*. For more information on user-defined functions and compiled modules, refer to Chapter 10, “Creating User-Defined Functions” and Chapter 11, “Employing User-Defined Functions in Tests” in the *Mercury WinRunner Advanced Features User’s Guide*.

To define a startup function:

- 1** Choose **File > Test Properties** to open the Test Properties dialog box.
- 2** Click the **Run** tab.



- 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. It cannot begin with a number or contain parentheses.

- 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.

Note: If both the calling test and the compiled module are saved in Quality Center, you must use the full path when calling the function.

- 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, refer to Chapter 11, “Employing User-Defined Functions in Tests” in the *Mercury WinRunner Advanced Features User’s Guide*.

23

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
- ▶ Setting Global Testing Options from the General Options Dialog Box
- ▶ Setting General Options
- ▶ Setting Folder Options
- ▶ Setting Recording Options
- ▶ Setting Test Run Options
- ▶ Setting Notification Options
- ▶ Setting Appearance Options
- ▶ Choosing Appropriate Timeout and Delay Settings

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 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, refer to Chapter 21, “Setting Testing Options from a Test Script” in the *Mercury WinRunner Advanced Features User’s Guide*.

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 subcategory in the options tree displays the corresponding options in the options pane.

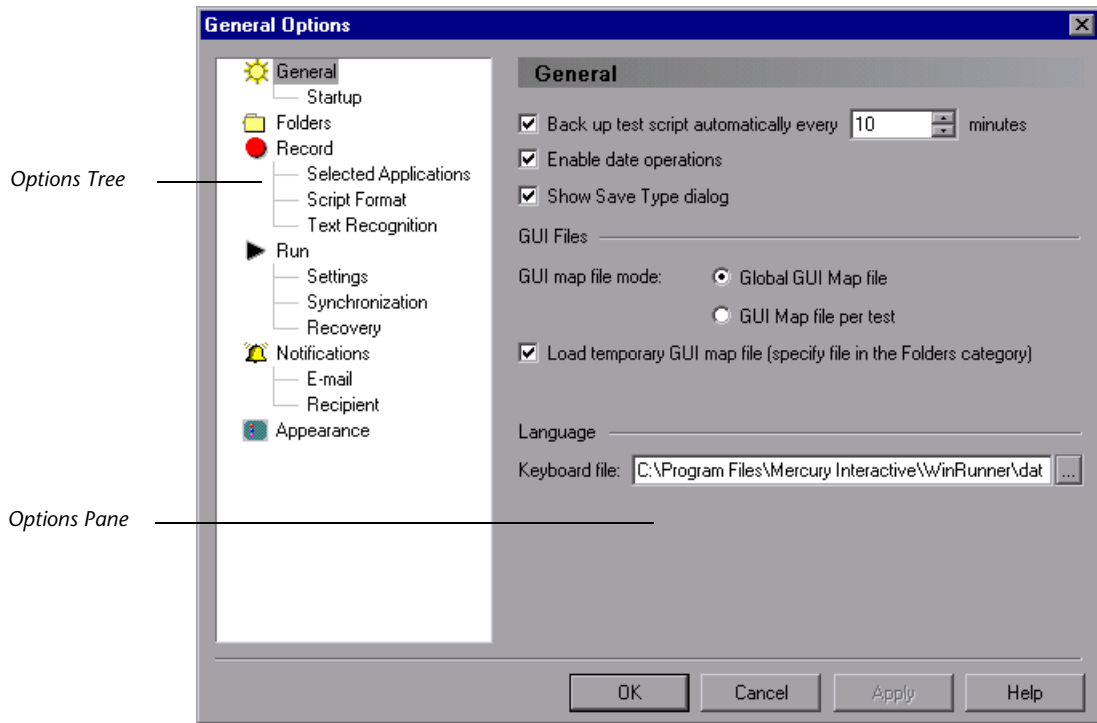
The General Options dialog box contains the following categories and subcategories:

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:

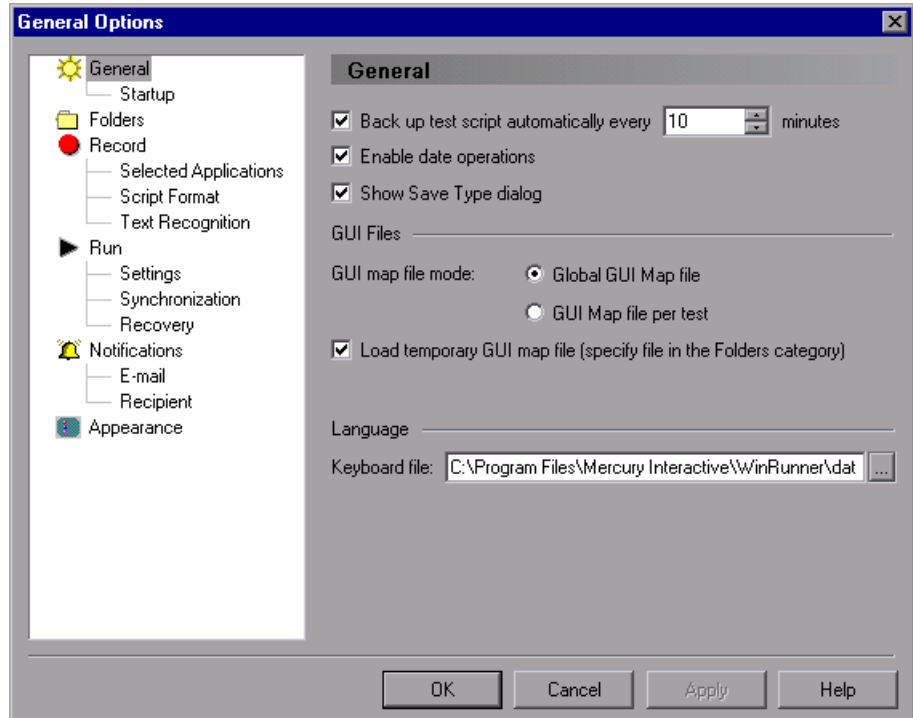
- 1 Choose **Tools > General Options**. The General Options dialog box opens.



- 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.



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:

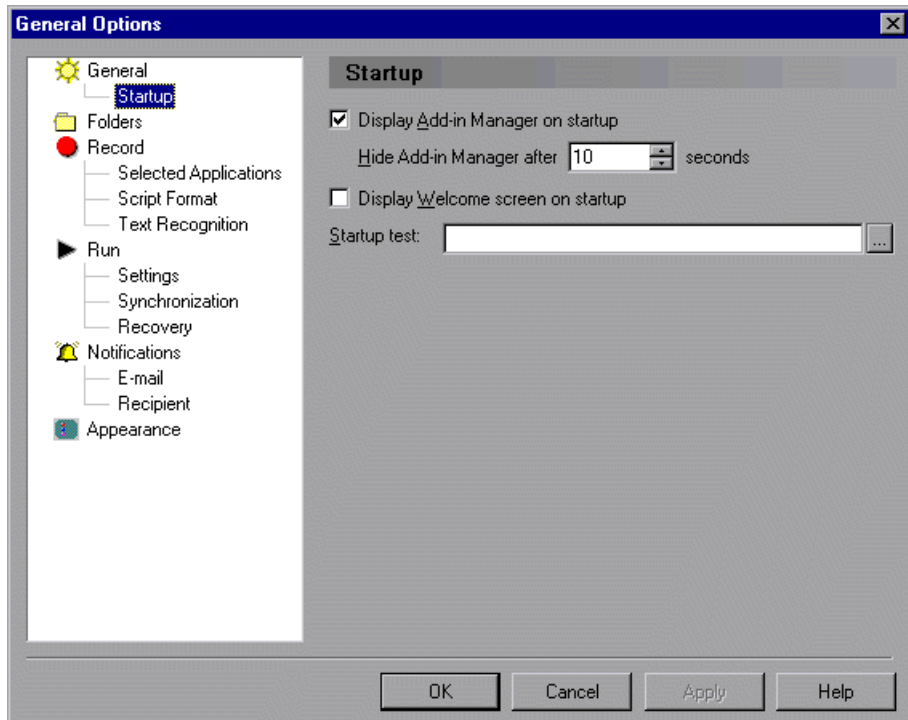
Option	Description
<p>Back up test script automatically every __ minutes</p>	<p>Instructs WinRunner to create a backup file for your script periodically, according to the specified interval. When selected, WinRunner creates a backup file in your test folder called script.sav, which is a simple text file of the script. Each time WinRunner backs up your script, it overwrites the previous script.sav file.</p> <p>Default = Selected, 10 [minutes]</p>
<p>Enable date operations</p>	<p>Enables date operation functionality and displays the Tools > Date menu item.</p> <p>Note: You must restart WinRunner for a change in this setting to take effect.</p> <p>Default = Cleared</p>
<p>Show Save Type dialog</p>	<p>This option is displayed only when WinRunner is connected to a Quality Center server.</p> <p>Displays the Select Type dialog box that enables you to save a new script as a WinRunner test or as a scripted component.</p> <p>Note: Clearing the Don't show it again check box at the bottom of the Select Type dialog box also clears the selection in the General pane.</p> <p>Default = Selected</p>

Option	Description
GUI map file mode	<p>Sets the GUI map file mode in WinRunner.</p> <ul style="list-style-type: none"> • 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.” <p>Note: You must restart WinRunner for a change in this setting to take effect.</p> <p>If you are working with tests created in WinRunner 6.02 or earlier, you must work in the <i>Global GUI Map File</i> mode.</p> <p>Default = Global GUI Map File</p>
Load temporary GUI map file	<p>Automatically loads the temporary GUI map file when starting WinRunner.</p> <p>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.</p> <p>You can set the location of the temporary GUI map file in the Folders category of the General Options dialog box.</p> <p>Default = selected</p>

Option	Description
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.



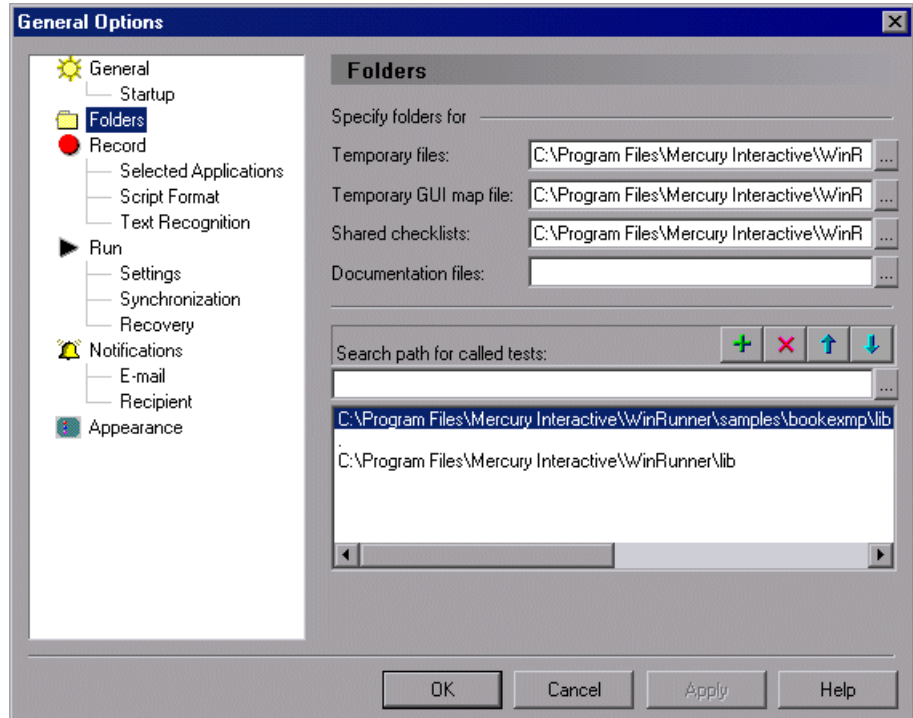
The **Startup** category contains the following options:

Option	Description
Display Add-in Manager on startup	<p>Displays the Add-In Manager dialog box when starting WinRunner.</p> <p>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.</p> <p>Default = Selected</p>
Hide Add-in Manager after ___ seconds	<p>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.</p> <p>Default = 10 seconds</p>

Option	Description
<p>Display Welcome screen on startup</p>	<p>Displays the Welcome screen when starting WinRunner.</p> <p>Note: Clearing the Show on Startup check box at the bottom of the Welcome screen also clears the selection in the Startup pane.</p> <p>Default = Selected</p>
<p>Startup test</p>	<p>Specifies the location of your startup test.</p> <p>You can use a startup test to perform operations such as configuring recording, loading compiled modules, and loading GUI map files when starting WinRunner.</p> <p>For more information, refer to Chapter 23, “Initializing Special Configurations” in the <i>Mercury WinRunner Advanced Features User’s Guide</i>.</p> <p>Note: You can also set the location of your startup test from the RapidTest Script wizard.</p> <p>A startup test can be used in addition to (and not instead of) the initialization (<i>tslinit</i>) test.</p> <p>You can specify a Quality Center script as your startup test. If you do, ensure that Reconnect on startup is selected in the Quality Center Connection dialog box. For more information, refer to Chapter 26, “Managing the Testing Process” in the <i>Mercury WinRunner Advanced Features User’s Guide</i>.</p> <p>Default = <WinRunner installation folder></p>

Setting Folder Options





The **Folders** category enables you to specify the locations of WinRunner files and to specify search paths for resolving relative paths.



The **Folders** category contains the following options:

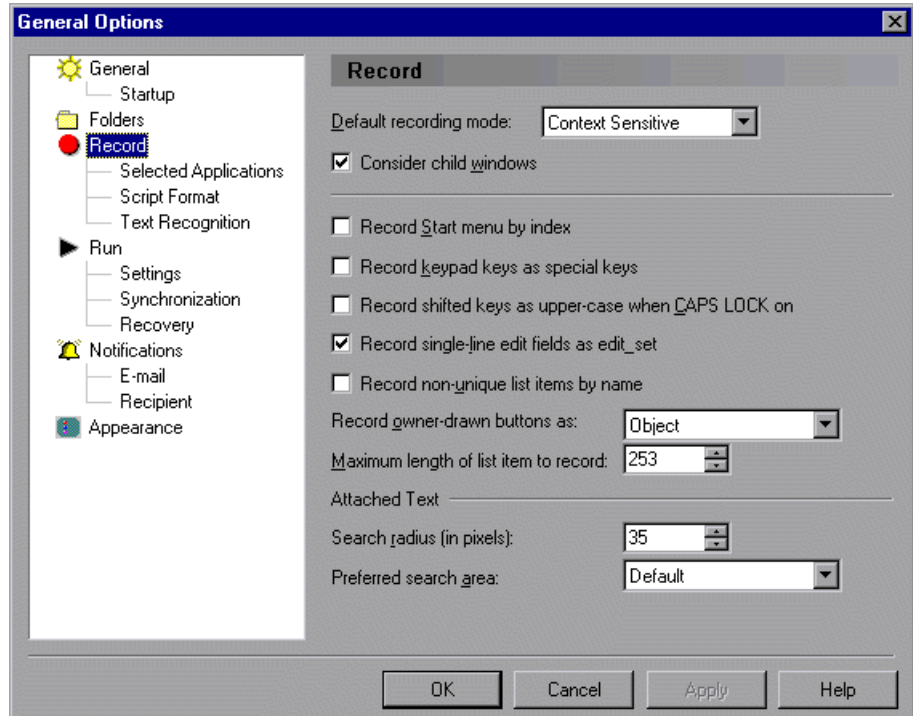
Option	Description
<p>Temporary files</p>	<p>The folder containing temporary tests. Enter or browse to the folder.</p> <p>Notes: If you designate a new folder, you must restart WinRunner in order for the change to take effect.</p> <p>You can use the setvar and getvar functions to set and retrieve the value of the corresponding <i>tempdir</i> testing option from within a test script, as described in Chapter 21, “Setting Testing Options from a Test Script” in the <i>Mercury WinRunner Advanced Features User’s Guide</i>.</p> <p>Default = <WinRunner installation folder>\tmp</p>
<p>Temporary GUI map file</p>	<p>The folder containing the temporary GUI map file (<i>temp.gui</i>). 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.</p> <p>Note: If you designate a new folder, you must restart WinRunner in order for the change to take effect.</p> <p>Default = <WinRunner installation folder>\tmp</p>

Option	Description
Shared checklists	<p>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 143. For more information on shared database checklists, see “Saving a Database Checklist in a Shared Folder” on page 297.</p> <p>Notes: If you designate a new folder, you must restart WinRunner in order for the change to take effect.</p> <p>You can use the getvar function to retrieve the value of the corresponding <i>shared_checklist_dir</i> testing option from within a test script, as described in Chapter 21, “Setting Testing Options from a Test Script” in the <i>Mercury WinRunner Advanced Features User’s Guide</i>.</p> <p>Default = <WinRunner installation folder>\chklist</p>
Documentation files	<p>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.</p> <p>Default = <WinRunner installation folder>\doc</p>

Option	Description
Search path for called tests	<p>The paths that WinRunner searches for files or tests specified with a relative path. If you define search paths in this pane, you can specify relative paths when calling tests and specifying other file names. The order of the search paths in the list determines the order of locations in which WinRunner searches for a file or test specified using a relative path.</p> <p>For more information, refer to Chapter 9, “Calling Tests” in the <i>Mercury WinRunner Advanced Features User’s Guide</i>.</p> <ul style="list-style-type: none"> • 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 . <p>When WinRunner is connected to Quality Center, you can specify the paths in a Quality Center database that WinRunner searches for called tests. Search paths in a Quality Center database are preceded by [QC]. Note that you cannot use the Browse button to specify search paths in a Quality Center database.</p> <p>Notes: You can use the setvar and getvar functions to set and retrieve the value of the corresponding <i>searchpath</i> testing option from within a test script, as described in Chapter 21, “Setting Testing Options from a Test Script” in the <i>Mercury WinRunner Advanced Features User’s Guide</i>. You can also set this option using the corresponding -search_path command line option, described in Chapter 15, “Running Tests from the Command Line” in the <i>Mercury WinRunner Advanced Features User’s Guide</i>.</p>

Setting Recording Options

The **Record** category contains options for controlling how WinRunner records tests.



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
<p>Default recording mode</p>	<p>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.”</p> <p>Default = Context Sensitive</p>
<p>Consider child windows</p>	<p>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.</p> <p>Note that you can use the setvar and getvar functions to set and retrieve the value of the corresponding <i>enum_descendent_toplevel</i> testing option from within a test script, as described in Chapter 21, “Setting Testing Options from a Test Script” in the <i>Mercury WinRunner Advanced Features User’s Guide</i>.</p> <p>Default = selected</p>

Option	Description
Record Start menu by index	<p>Determines how WinRunner records on the Windows Start menu in Windows NT.</p> <p>When this option is selected, WinRunner records the index IDs for each selected menu item. For example:</p> <pre>button_press ("Start"); menu_select_item ("item_2;item_0;item_4");</pre> <p>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.</p> <p>When this option is cleared, WinRunner records the name of the menu items in the start menu. For example:</p> <pre>button_press ("Start"); menu_select_item ("Programs;Accessories;Calculator");</pre> <p>Default = cleared</p>

Option	Description
<p>Record keypad keys as special keys</p>	<p>Determines how WinRunner records pressing keys on the numeric keyboard.</p> <p>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:</p> <pre>obj_type ("Edit", "<kNumLock>") obj_type ("Edit", "<kKP7>")</pre> <p>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:</p> <pre>obj_type ("Edit", "7");</pre> <p>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.</p> <p>Default = cleared</p>
<p>Record shifted keys as uppercase when CAPS LOCK on</p>	<p>Determines whether WinRunner records pressing letter keys and the SHIFT key together as uppercase letters when CAPS LOCK is activated.</p> <p>When this option is selected, WinRunner records pressing letter keys and the SHIFT key together as uppercase letters even when CAPS LOCK is activated. Therefore, WinRunner ignores the state of the CAPS LOCK key when recording and running tests.</p> <p>When this option is cleared, WinRunner records pressing letter keys and the SHIFT key together as lowercase letters when CAPS LOCK is activated.</p> <p>Default = cleared</p>

Option	Description
Record single-line edit fields as edit_set	<p>Determines how WinRunner records typing a string in a single-line edit field.</p> <p>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 box in the Flights Reservation application, you type H, press BACKSPACE, and then type Jennifer, WinRunner generates the following statement:</p> <pre>edit_set ("Name", "Jennifer");</pre> <p>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:</p> <pre>obj_type ("Name", "H<kBackSpace>Jennifer");</pre> <p>For more information about the edit_set and obj_type functions, refer to the <i>TSL Reference</i>.</p> <p>Default = selected</p>
Record non-unique list items by name	<p>Determines whether WinRunner records non-unique list box and combo box items by name (selected) or by index (cleared).</p> <p>Note: You can use the setvar and getvar functions to set and retrieve the value of the corresponding <i>rec_item_name</i> testing option from within a test script, as described in Chapter 21, “Setting Testing Options from a Test Script” in the <i>Mercury WinRunner Advanced Features User’s Guide</i>. You can also set this option using the corresponding -rec_item_name command line option, described in Chapter 15, “Running Tests from the Command Line” in the <i>Mercury WinRunner Advanced Features User’s Guide</i>.</p> <p>Default = cleared</p>

Option	Description
Record owner-drawn buttons as	<p>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).</p> <p>Note that you can use the setvar and getvar functions to set and retrieve the value of the corresponding <i>rec_owner_drawn</i> testing option from within a test script, as described in Chapter 21, “Setting Testing Options from a Test Script” in the <i>Mercury WinRunner Advanced Features User’s Guide</i>.</p> <p>Default = Object</p>
Maximum length of list item to record	<p>Defines the maximum number of characters that WinRunner can record in a list item name.</p> <p>If the maximum number of characters is exceeded in a list view or tree view item, WinRunner records that item’s index number.</p> <p>If the maximum number of characters is exceeded in a list box or combo box, WinRunner truncates the item’s name. The maximum length can be 1 to 253 characters.</p> <p>Default = 253 [characters]</p>
Attached Text	<p>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.</p> <p>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.</p> <p>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.</p>

Option	Description
Search radius	<p>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.</p> <p>You can use the setvar and getvar functions to set and retrieve the value of the corresponding <i>attached_text_search_radius</i> testing option from within a test script, as described in Chapter 21, “Setting Testing Options from a Test Script” in the <i>Mercury WinRunner Advanced Features User’s Guide</i>.</p> <p>Default= 35 [pixels]</p>
Preferred search area	<p>Specifies the location on a GUI object from which WinRunner searches for its attached text.</p> <p>You can use the setvar and getvar functions to set and retrieve the value of the corresponding <i>attached_text_area</i> testing option from within a test script, as described in Chapter 21, “Setting Testing Options from a Test Script” in the <i>Mercury WinRunner Advanced Features User’s Guide</i>.</p> <p>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.</p> <p>Default = Default</p>

Setting Selected Applications

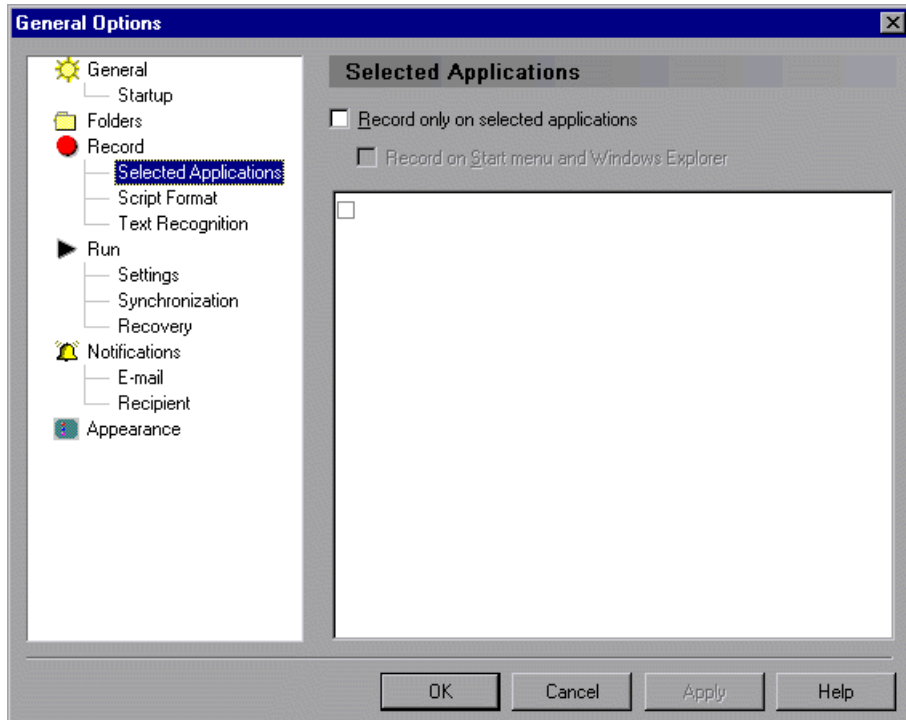
The Selected Applications pane enables you to instruct WinRunner to record your operations on selected programs while ignoring operations on other programs. For example, you may not want to record operations you perform on your e-mail client while recording a test.

When you enable selective recording, only actions on the selected programs are recorded.

Note that even if you choose to record only on selected applications, you can still create checkpoints and perform all other non-recording 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.

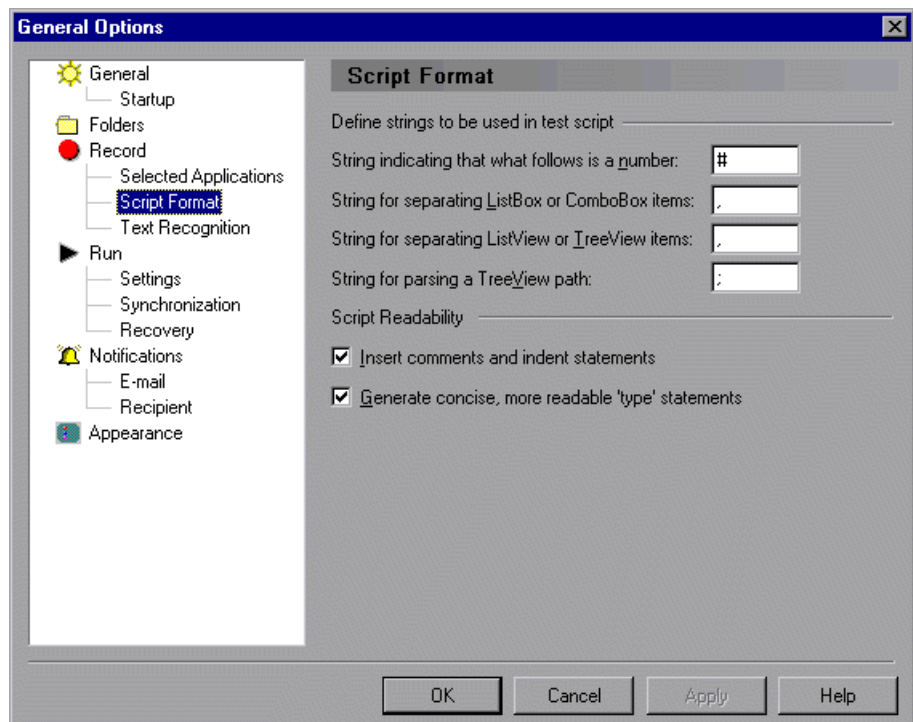


- 3 Select **Record only on selected applications**.
- 4 If you want to record operations on the **Start** menu and on Windows Explorer, select **Record on Start menu and Windows Explorer**. The relevant files are automatically added to the list.
- 5 If you do not want to record on Internet Explorer and/or Netscape, clear the options for **ieexplore.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.



The **Script Format** category contains the following options:

Option	Description
<p>String indicating that what follows is a number</p>	<p>The string recorded in the test script to indicate that a list item is specified by its index number.</p> <p>Note that you can use the setvar and getvar functions to set and retrieve the value of the corresponding <i>item_number_seq</i> testing option from within a test script, as described in Chapter 21, “Setting Testing Options from a Test Script” in the <i>Mercury WinRunner Advanced Features User’s Guide</i>.</p> <p>Default = #</p>
<p>String for separating ListBox or ComboBox items</p>	<p>The string recorded in the test script to separate items in a list box or a combo box.</p> <p>You can use the setvar and getvar functions to set and retrieve the value of the corresponding <i>list_item_separator</i> testing option from within a test script, as described in Chapter 21, “Setting Testing Options from a Test Script” in the <i>Mercury WinRunner Advanced Features User’s Guide</i>.</p> <p>Default = ,</p>
<p>String for separating ListView or TreeView items</p>	<p>The string recorded in the test script to separate items in a list view or a tree view.</p> <p>You can use the setvar and getvar functions to set and retrieve the value of the corresponding <i>listview_item_separator</i> testing option from within a test script, as described in Chapter 21, “Setting Testing Options from a Test Script” in the <i>Mercury WinRunner Advanced Features User’s Guide</i>.</p> <p>Default = ,</p>

Option	Description
String for parsing a TreeView path	<p>The string recorded in the test script to separate items in a tree view path.</p> <p>You can use the setvar and getvar functions to set and retrieve the value of the corresponding <i>treeview_path_separator</i> testing option from within a test script, as described in Chapter 21, “Setting Testing Options from a Test Script” in the <i>Mercury WinRunner Advanced Features User’s Guide</i>.</p> <p>Default = ;</p>
Insert comments and indent statements	<p>Determines whether WinRunner automatically divides your test script into sections while you record.</p> <p>For more information, see “Inserting Comments and Indent Statements” below.</p> <p>Default = selected</p>

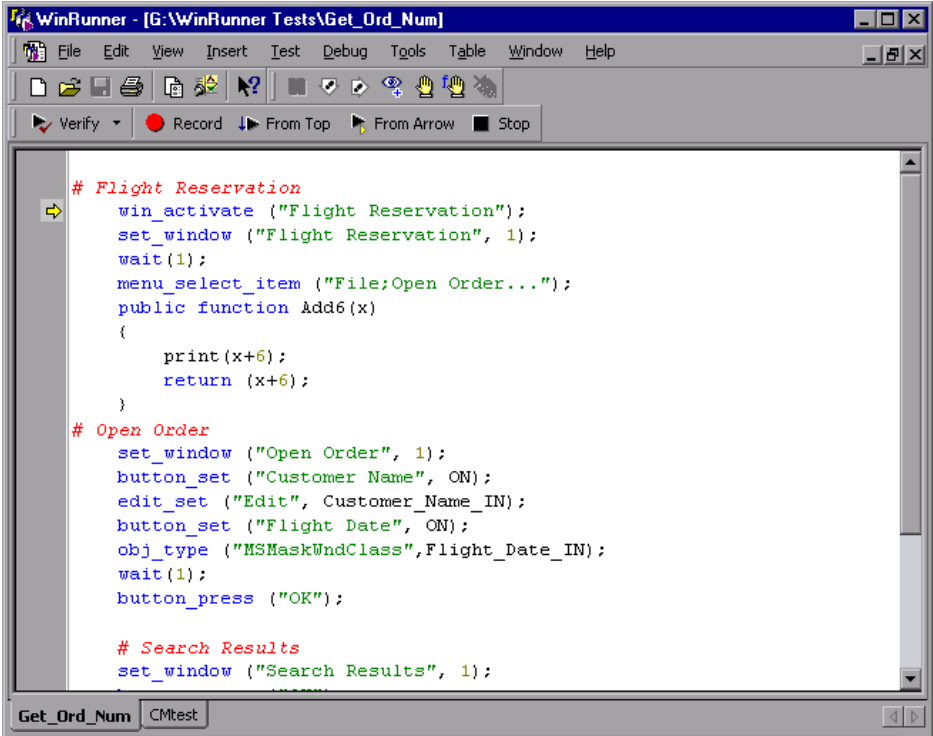
Option	Description
<p>Generate concise, more readable 'type' statements</p>	<p>Determines how WinRunner generates type, win_type, and obj_type statements in a test script.</p> <p>When this option is selected, WinRunner generates more concise type, win_type, and obj_type statements that represent only the net result of pressing and releasing input keys. This makes your test script easier to read. For example:</p> <pre>obj_type (object, "A");</pre> <p>When this option is cleared, WinRunner records the pressing and releasing of each key. For example: <pre>obj_type (object, "<kShift_L>-a-a+<kShift_L>+");</pre> <p>Clear this option if the exact order of keystrokes is important for your test.</p> <p>For more information, refer to the type, win_type, and obj_type functions in the <i>TSL Reference</i>.</p> <p>You can use the setvar and getvar functions to set and retrieve the value of the corresponding <i>key_editing</i> testing option from within a test script, as described in Chapter 21, "Setting Testing Options from a Test Script" in the <i>Mercury WinRunner Advanced Features User's Guide</i>.</p> <p>Default = selected</p> </p>

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 group all statements related to the same window.



```

WinRunner - [G:\WinRunner Tests\Get_Ord_Num]
File Edit View Insert Test Debug Tools Table Window Help
Verify Record From Top From Arrow Stop

# Flight Reservation
win_activate ("Flight Reservation");
set_window ("Flight Reservation", 1);
wait(1);
menu_select_item ("File;Open Order...");
public function Add6(x)
{
    print(x+6);
    return (x+6);
}

# Open Order
set_window ("Open Order", 1);
button_set ("Customer Name", ON);
edit_set ("Edit", Customer_Name_IN);
button_set ("Flight Date", ON);
obj_type ("MSMaskWndClass", Flight_Date_IN);
wait(1);
button_press ("OK");

# Search Results
set_window ("Search Results", 1);

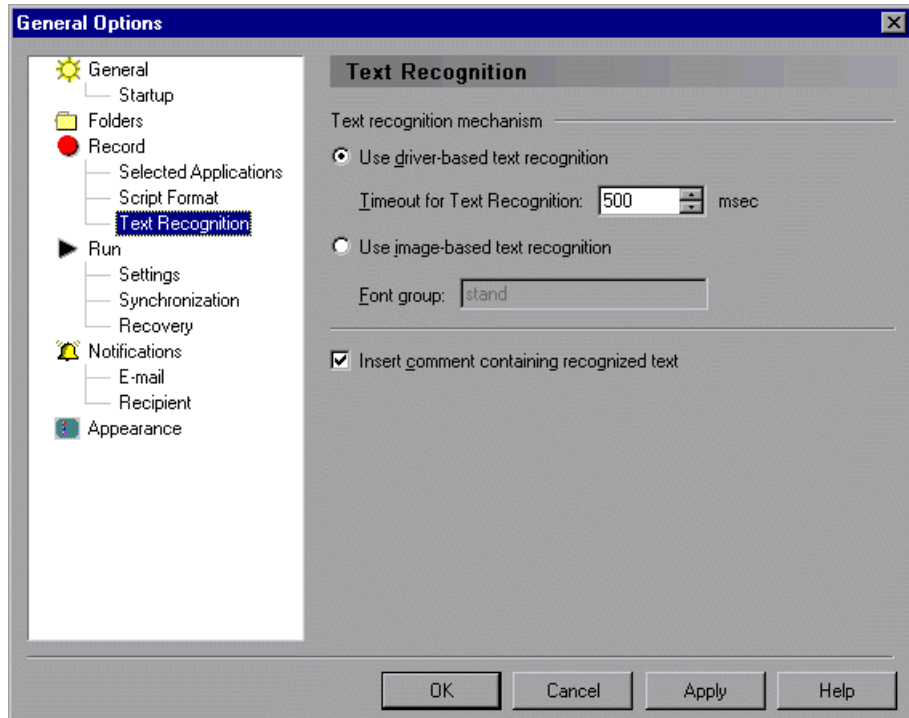
```

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.



The **Text Recognition** category contains the following options:

Option	Description
Use driver-based text recognition	<p>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 you are testing, select Use image-based text recognition.</p> <p>Default = selected</p>
Timeout for Text Recognition	<p>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.</p> <p>See “Choosing Appropriate Timeout and Delay Settings” on page 589 for more information on when to adjust this setting.</p> <p>Default = 500 [milliseconds]</p>
Use image-based text recognition	<p>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.</p> <p>Default = cleared</p>

Option	Description
Font group	<p>Sets the active font group for image text recognition. For more information on font groups, see “Teaching Fonts to WinRunner” on page 342.</p> <p>You can use the setvar and getvar functions to set and retrieve the value of the corresponding <i>fontgrp</i> testing option from within a test script, as described in Chapter 21, “Setting Testing Options from a Test Script” in the <i>Mercury WinRunner Advanced Features User’s Guide</i>.</p> <p>You can also set this option using the corresponding -fontgrp command line option, described in Chapter 15, “Running Tests from the Command Line” in the <i>Mercury WinRunner Advanced Features User’s Guide</i>.</p> <p>Default = stand</p>
Insert comment containing recognized text	<p>When you create a text checkpoint, this option determines how WinRunner displays the captured text in the test script.</p> <p>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:</p> <pre>obj_get_text("Fly From:", text);# Portland</pre> <p>Default = selected</p>

Considerations for Using Text Recognition for Windows-Based Applications

You use the WinRunner text recognition mechanism when:

- Inserting text checkpoints using **Insert > Get Text > From Screen Area** and **Insert > Get Text > From Object/Window**
- Retrieving or checking the **text** property of GUI objects, using functions ending with **_get_info** or **_check_info**

- Retrieving or checking text using functions ending with `_get_text` or `_check_text`
- Performing other text-based operations using functions ending with `_find_text`, `_move_locator_text`, or `_click_on_text` functions

When using the WinRunner text-recognition mechanism for Windows-based applications, keep in mind that it may occasionally retrieve unwanted text information (such as hidden text and shadowed text, which appears as multiple copies of the same string).

Additionally, the text recognition may behave differently in different run sessions depending on the operating system version you are using, service packs you have installed, other installed toolkits, the APIs used in your application, and so on.

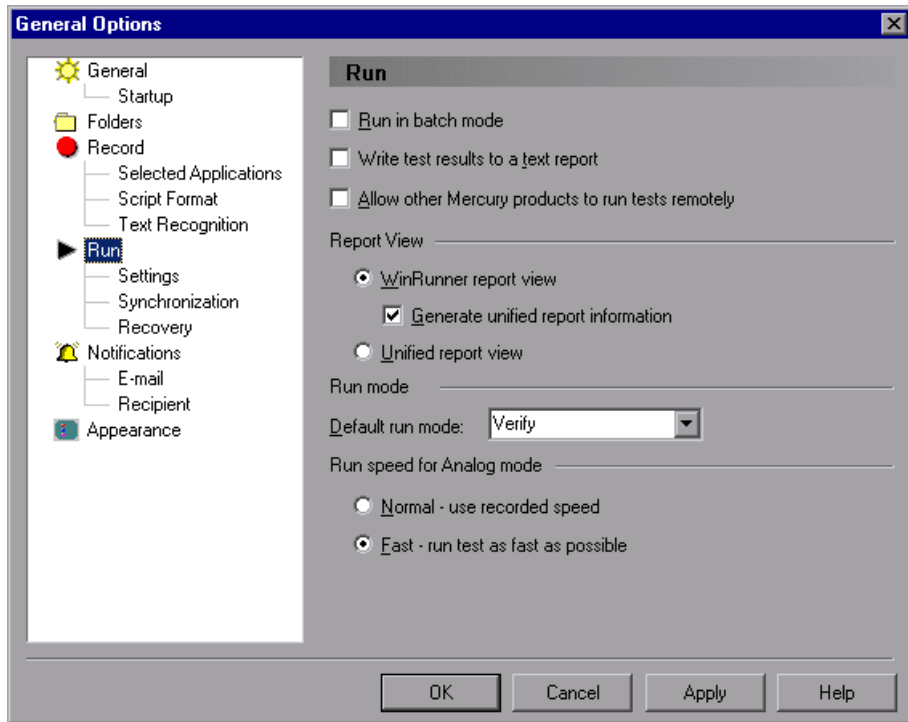
Therefore, when possible, it is highly recommended to retrieve or check text from your application window by inserting a standard GUI checkpoint and selecting to check the object's **value** (or similar) property. For example:

- Instead of choosing **Insert > Get Text > From Object/Window**, choose **Insert > GUI Checkpoint > For Single Property** and select to check the **value** property.
- Instead of `edit_get_text("Edit", result);` or `edit_get_info("Edit", "text", result);`, use `edit_get_info("Edit", "value", result);`
- Instead of `edit_check_text("Edit", exp_val);` or `edit_check_info("Edit", "text", exp_val);`, use `edit_check_info("Edit", "value", expected_result);`

Note: The above issues do not apply when working with Web-based applications.

Setting Test Run Options

The **Run** category options control how WinRunner runs tests.



In addition to the options in this category, you can set additional recording options in the **Settings**, **Synchronization**, and **Recovery** subcategories.

The **Run** category contains the following options:

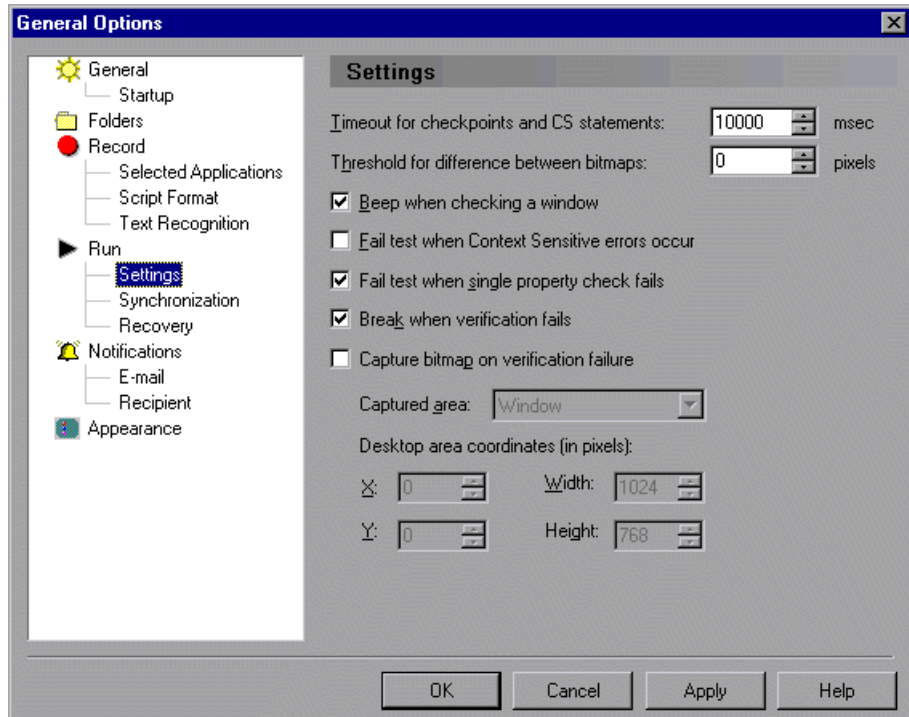
Option	Description
Run in batch mode	<p>Determines whether WinRunner suppresses messages during a Verify test run so that a test can run unattended.</p> <p>For example, if a set_window statement is missing from a test script, WinRunner cannot find the specified window. If the test runs in batch mode, 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 you to locate the window.</p> <p>Note: Messages are suppressed for a batch test only if you run the test using the Verify run mode. If you use the Update or Debug run mode to run the test, some messages may be displayed even when the Run in batch mode option is selected.</p> <p>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.</p> <p>For more information on suppressing messages during a test run, refer to Chapter 14, “Running Batch Tests” in the <i>Mercury WinRunner Advanced Features User’s Guide</i>.</p> <p>You can use the getvar function to retrieve the value of the corresponding <i>batch</i> testing option from within a test script, as described in Chapter 21, “Setting Testing Options from a Test Script” in the <i>Mercury WinRunner Advanced Features User’s Guide</i>.</p> <p>You can also set this option using the corresponding -batch command line option, described in Chapter 15, “Running Tests from the Command Line” in the <i>Mercury WinRunner Advanced Features User’s Guide</i>.</p> <p>Default = cleared</p>

Option	Description
<p>Write test results to a text report</p>	<p>Instructs WinRunner to automatically write test results to a text report, called report.txt, which is saved in the results folder.</p> <p>You can also set this option using the corresponding -create_text_report command line option, described in Chapter 15, “Running Tests from the Command Line” in the <i>Mercury WinRunner Advanced Features User’s Guide</i>.</p> <p>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.</p> <p>Default = cleared</p>
<p>Allow other Mercury products to run tests remotely</p>	<p>Enables other Mercury products to run WinRunner tests on your computer from a remote machine.</p> <p>For information on running WinRunner tests remotely from other Mercury products, refer to the documentation for those products.</p>
<p>WinRunner report view</p>	<p>Displays the test results using the WinRunner test results display.</p> <p>Default = selected</p>
<p>Generate unified report information</p>	<p>Generates the necessary information for creating a unified report so that you can choose to view the results of your tests in the unified report view at a later time.</p> <p>(Enabled only when WinRunner report view is selected.)</p> <p>Default = selected</p>
<p>Unified report view</p>	<p>Generates unified report information during the test run and displays the test results using the unified report design. This display enables you to view all WinRunner events and QuickTest steps in a single report.</p> <p>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.</p> <p>Default = cleared</p>

Option	Description
Default run mode	<p>Enables you to select the run mode that is used by default for all tests.</p> <ul style="list-style-type: none"> • Update—Used to update the expected results of a test or to create a new expected results folder. • Verify—Used to test your application. • Debug—Used to help you identify bugs in a test script. <p>Note: Verify mode is only relevant when running tests, not components. When working with components, the application is verified when the component is run as part of a business process test in Quality Center.</p> <p>For more information on run modes, see “WinRunner Test Run Modes” on page 429.</p> <p>Default = Verify</p>
Run speed for Analog mode	<p>Determines the default run speed for tests run in Analog mode.</p> <p>Normal—runs the test at the speed at which it was recorded.</p> <p>Fast—runs the test as fast as the application can receive input.</p> <p>You can use the setvar and getvar functions to set and retrieve the value of the corresponding <i>speed</i> testing option from within a test script, as described in Chapter 21, “Setting Testing Options from a Test Script” in the <i>Mercury WinRunner Advanced Features User’s Guide</i>.</p> <p>You can also set this option using the corresponding -speed command line option, described in Chapter 15, “Running Tests from the Command Line” in the <i>Mercury WinRunner Advanced Features User’s Guide</i>.</p> <p>Default = Fast</p>

Setting Run Setting Options

The **Settings** category contains options for handling specific situations during the test run.



The **Settings** category contains the following options:

Option	Description
Timeout for checkpoints and CS statements	<p>Sets the global timeout (in milliseconds) that WinRunner uses when performing checkpoints and Context Sensitive statements. This value is added to the <i>time</i> 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).</p> <p>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.</p> <p>Note: This option is accurate to within 20-30 milliseconds.</p> <p>See “Choosing Appropriate Timeout and Delay Settings” on page 589 for more information on when to adjust this setting.</p> <p>You can use the setvar and getvar functions to set and retrieve the value of the corresponding <i>timeout_msec</i> testing option from within a test script, as described in Chapter 21, “Setting Testing Options from a Test Script” in the <i>Mercury WinRunner Advanced Features User’s Guide</i>.</p> <p>You can also set this option using the corresponding -timeout_msec command line option, described in Chapter 15, “Running Tests from the Command Line” in the <i>Mercury WinRunner Advanced Features User’s Guide</i>.</p> <p>Default = 10000 [milliseconds]</p>

Option	Description
Threshold for difference between bitmaps	<p>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.</p> <p>You can use the setvar and getvar functions to set and retrieve the value of the corresponding <i>min_diff</i> testing option from within a test script, as described in Chapter 21, “Setting Testing Options from a Test Script” in the <i>Mercury WinRunner Advanced Features User’s Guide</i>.</p> <p>You can also set this option using the corresponding -min_diff command line option, described in Chapter 15, “Running Tests from the Command Line” in the <i>Mercury WinRunner Advanced Features User’s Guide</i>.</p> <p>Default = 0 (pixels)</p>
Beep when checking a window	<p>Determines whether WinRunner beeps when checking any window during a test run.</p> <p>Note that you can use the setvar and getvar functions to set and retrieve the value of the corresponding <i>beep</i> testing option from within a test script, as described in Chapter 21, “Setting Testing Options from a Test Script” in the <i>Mercury WinRunner Advanced Features User’s Guide</i>.</p> <p>Note that you can also set this option using the corresponding -beep command line option, described in Chapter 15, “Running Tests from the Command Line” in the <i>Mercury WinRunner Advanced Features User’s Guide</i>.</p> <p>Default = selected</p>

Option	Description
Fail test when Context Sensitive errors occur	<p>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.</p> <p>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 <i>TSL Reference</i>.</p> <p>You can use the setvar and getvar functions to set and retrieve the value of the corresponding <i>cs_fail</i> testing option from within a test script, as described in Chapter 21, “Setting Testing Options from a Test Script” in the <i>Mercury WinRunner Advanced Features User’s Guide</i>.</p> <p>You can also set this option using the corresponding -cs_fail command line option, described in Chapter 15, “Running Tests from the Command Line” in the <i>Mercury WinRunner Advanced Features User’s Guide</i>.</p> <p>Default = cleared</p>

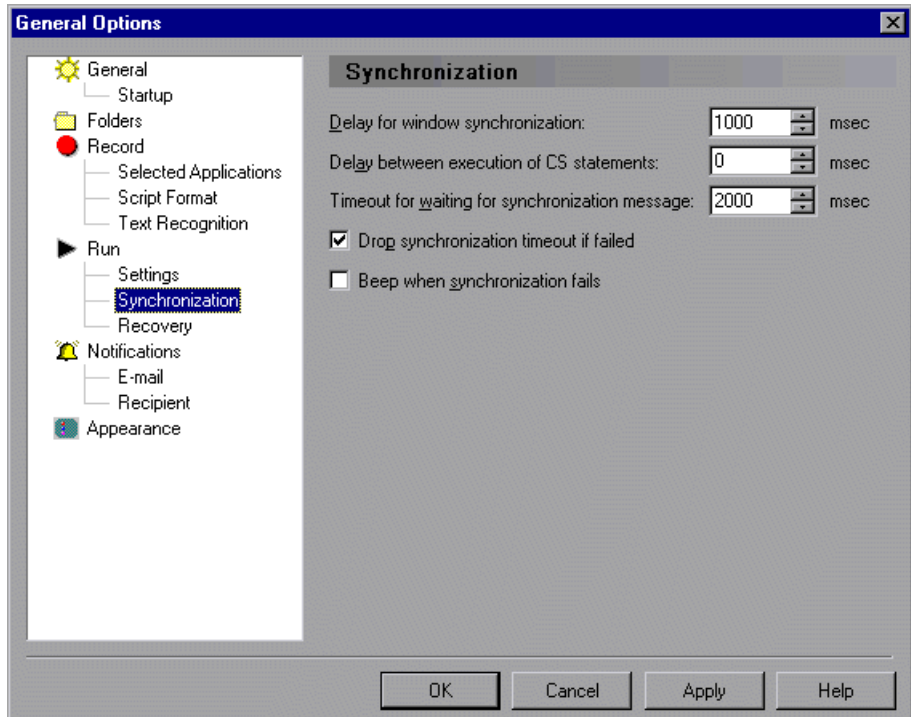
Option	Description
<p>Fail test when single property check fails</p>	<p>Determines whether WinRunner fails a test when _check_info statements fail. It also writes an event to the Test Results window for these statements.</p> <p>(You can create _check_info statements using the Insert > GUI Checkpoint > For Single Property command.)</p> <p>For information about the check_info functions, refer to the <i>TSL Reference</i>.</p> <p>You can use the setvar and getvar functions to set and retrieve the value of the corresponding <i>single_prop_check_fail</i> testing option from within a test script, as described in Chapter 21, “Setting Testing Options from a Test Script” in the <i>Mercury WinRunner Advanced Features User’s Guide</i>.</p> <p>You can also set this option using the corresponding -single_prop_check_fail command line option, described in Chapter 15, “Running Tests from the Command Line” in the <i>Mercury WinRunner Advanced Features User’s Guide</i>.</p> <p>Default = selected</p>

Option	Description
Break when verification fails	<p>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).</p> <p>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 you 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.</p> <p>You can use the setvar and getvar functions to set and retrieve the value of the corresponding <i>mismatch_break</i> testing option from within a test script, as described in Chapter 21, “Setting Testing Options from a Test Script” in the <i>Mercury WinRunner Advanced Features User’s Guide</i>.</p> <p>You can also set this option using the corresponding -mismatch_break command line option, described in Chapter 15, “Running Tests from the Command Line” in the <i>Mercury WinRunner Advanced Features User’s Guide</i>.</p> <p>Default = selected</p>
Capture bitmap on verification failure	<p>Instructs WinRunner to capture an image of your application each time a checkpoint fails. The bitmap is saved in your test results folder.</p> <p>Default = cleared</p> <p>You can use the setvar and getvar functions to set and retrieve the value of the corresponding capture_bitmap testing option from within a test script, as described in Chapter 21, “Setting Testing Options from a Test Script” in the <i>Mercury WinRunner Advanced Features User’s Guide</i>.</p>

Option	Description
Captured area	<p>Specifies the area of your screen to capture when a checkpoint fails.</p> <p>Window—Captures the active window.</p> <p>Desktop—Captures the entire desktop.</p> <p>Desktop Area—Captures the specified area of the desktop.</p>
Desktop area coordinates	<p>X—The x-coordinate of the top, left corner of the rectangle area to capture.</p> <p>Y—The y-coordinate of the top, left corner of the rectangle area to capture.</p> <p>Width—The width of the rectangle to capture.</p> <p>Height—The height of the rectangle to capture.</p> <p>(Enabled only when Desktop Area is the selected Captured area.)</p>

Setting Run Synchronization Options

The Synchronization category defines synchronization settings for your test run.



The **Synchronization** category contains the following options:

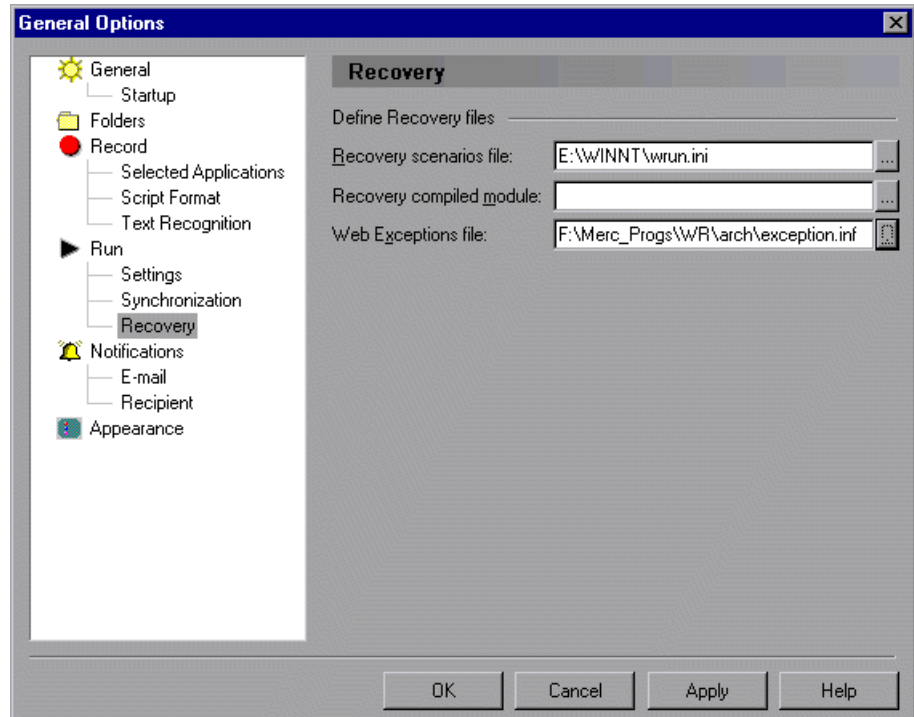
Option	Description
<p>Delay for window synchronization</p>	<p>Sets the sampling interval (in milliseconds) used to determine that a window is stable before capturing it for a Context Sensitive checkpoint or synchronization point. To be declared stable, a window must not change between two consecutive 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.</p> <p>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.</p> <p>This option is accurate to within 20-30 milliseconds.</p> <p>See “Choosing Appropriate Timeout and Delay Settings” on page 589 for more information on when to adjust this setting.</p> <p>You can use the setvar and getvar functions to set and retrieve the value of the corresponding <i>delay_msec</i> testing option from within a test script, as described in Chapter 21, “Setting Testing Options from a Test Script” in the <i>Mercury WinRunner Advanced Features User’s Guide</i>.</p> <p>You can also set this option using the corresponding -delay_msec command line option, described in Chapter 15, “Running Tests from the Command Line” in the <i>Mercury WinRunner Advanced Features User’s Guide</i>.</p> <p>Default = 1000 [milliseconds]</p>

Option	Description
<p>Delay between execution of CS statements</p>	<p>Sets the time (in milliseconds) that WinRunner waits before executing each Context Sensitive statement when running a test.</p> <p>See “Choosing Appropriate Timeout and Delay Settings” on page 589 for more information on when to adjust this setting.</p> <p>You can use the setvar and getvar functions to set and retrieve the value of the corresponding <i>cs_run_delay</i> testing option from within a test script, as described in Chapter 21, “Setting Testing Options from a Test Script” in the <i>Mercury WinRunner Advanced Features User’s Guide</i>.</p> <p>You can also set this option using the corresponding -cs_run_delay command line option, described in Chapter 15, “Running Tests from the Command Line” in the <i>Mercury WinRunner Advanced Features User’s Guide</i>.</p> <p>Default = 0 [milliseconds]</p>
<p>Timeout for waiting for synchronization message</p>	<p>Sets the timeout (in milliseconds) that WinRunner waits before validating that keyboard or mouse input was entered correctly during a test run.</p> <p>If synchronization often fails during your test runs, consider increasing the value of this option.</p> <p>See “Choosing Appropriate Timeout and Delay Settings” on page 589 for more information on when to adjust this setting.</p> <p>You can use the setvar and getvar functions to set and retrieve the value of the corresponding <i>synchronization_timeout</i> testing option from within a test script, as described in Chapter 21, “Setting Testing Options from a Test Script” in the <i>Mercury WinRunner Advanced Features User’s Guide</i>.</p> <p>Default = 2000 [milliseconds]</p>

Option	Description
Drop synchronization timeout if failed	<p>Determines whether WinRunner minimizes the synchronization timeout (as defined in the Timeout for Waiting for Synchronization Message option above) after the first synchronization failure.</p> <p>See “Choosing Appropriate Timeout and Delay Settings” on page 589 for more information on when to adjust this setting.</p> <p>You can use the setvar and getvar functions to set and retrieve the value of the corresponding <i>drop_sync_timeout</i> testing option from within a test script, as described in Chapter 21, “Setting Testing Options from a Test Script” in the <i>Mercury WinRunner Advanced Features User’s Guide</i>.</p> <p>Default = selected</p>
Beep when synchronization fails	<p>Determines whether WinRunner beeps when the timeout for waiting for synchronization message fails.</p> <p>This option is primarily for debugging test scripts.</p> <p>If synchronization often fails during your test runs, consider increasing the value of the Timeout for Waiting for Synchronization Message option or the corresponding <i>synchronization_timeout</i> testing option with the setvar function from within a test script.</p> <p>See “Choosing Appropriate Timeout and Delay Settings” on page 589 for more information on when to adjust this setting.</p> <p>You can use the setvar and getvar functions to set and retrieve the value of the corresponding <i>sync_fail_beep</i> testing option from within a test script, as described in Chapter 21, “Setting Testing Options from a Test Script” in the <i>Mercury WinRunner Advanced Features User’s Guide</i>.</p> <p>Default = cleared</p>

Setting Recovery Options

The Recovery category options specify the files to which WinRunner refers for recovery scenario and Web exception information.

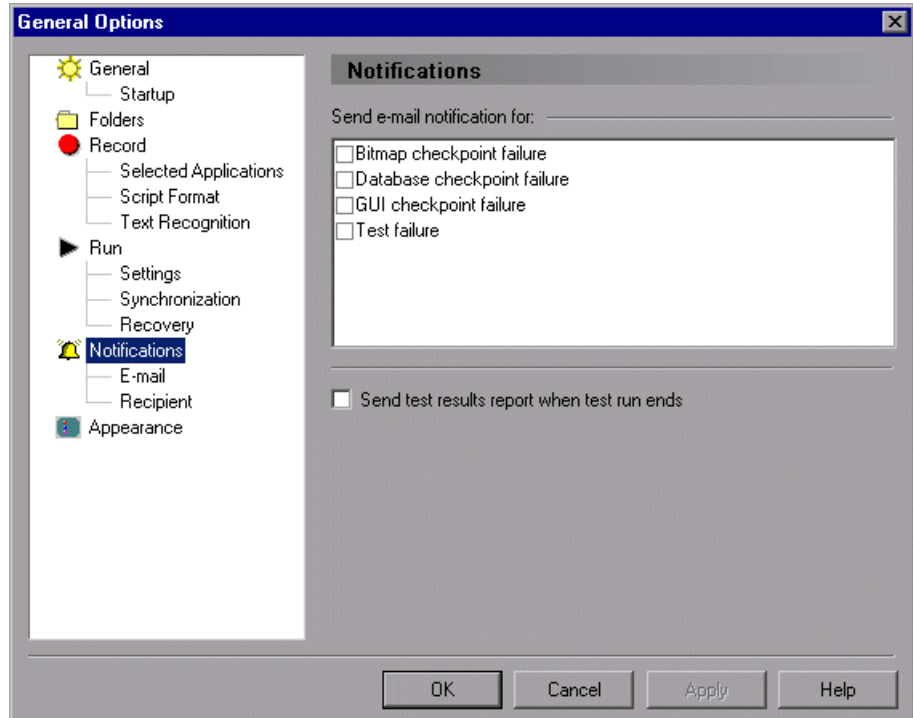


The **Recovery** category contains the following options:

Option	Description
<p>Recovery scenario file</p>	<p>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 <i>wrun.ini</i> before you can use the Recovery Manager to create or modify recovery scenarios.</p> <p>Recovery scenarios are defined and modified in the Recovery Manager. For more information, refer to Chapter 4, “Defining and Using Recovery Scenarios” in the <i>Mercury WinRunner Advanced Features User’s Guide</i>.</p> <p>Default = <Windows folder>\wrun.ini</p>
<p>Recovery compiled module</p>	<p>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, refer to Chapter 4, “Defining and Using Recovery Scenarios” in the <i>Mercury WinRunner Advanced Features User’s Guide</i>.</p> <p>Note: You can specify a Quality Center script as your recovery compiled module. If you do, ensure that Reconnect on startup is selected in the Quality Center Connection dialog box. For more information, refer to Chapter 26, “Managing the Testing Process” in the <i>Mercury WinRunner Advanced Features User’s Guide</i>.</p>
<p>Web Exceptions file</p>	<p>Indicates the location of the Web exceptions file, which stores the details of the available Web exception handling definitions.</p> <p>Web exceptions are defined and modified in the Web Exception Editor. For more information, refer to Chapter 5, “Handling Web Exceptions” in the <i>Mercury WinRunner Advanced Features User’s Guide</i>.</p> <p>Default = <WinRunner installation folder>\arch\exception.inf</p>

Setting Notification Options

The **Notifications** category contains options for sending e-mail notifications based on specified criteria.



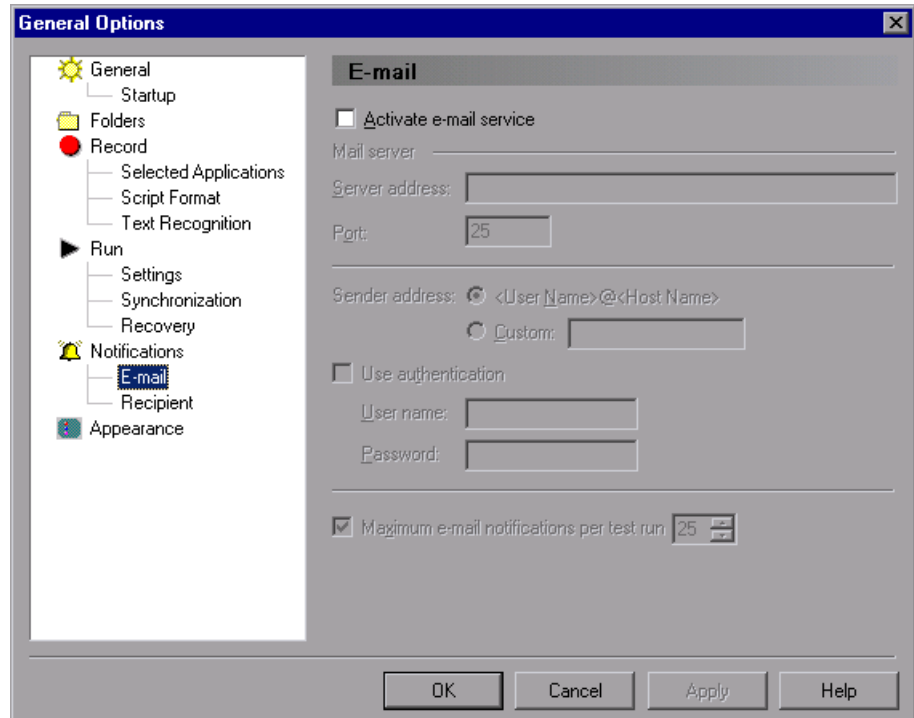
In addition to the options in this category, you can set additional notification options in the **E-mail** and **Recipient** subcategories.

The **Notifications** category contains the following options:

Option	Description
<p>Send e-mail notification for</p>	<p>Sends an e-mail for the selected conditions. You can select one or more of the following:</p> <ul style="list-style-type: none"> • 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. <p>For information on specifying recipients, see “Setting Notification Recipients Options” on page 584.</p> <p>Note: To enable the notification options, you must select to Activate e-mail service option in the E-mail category.</p> <p>Default = all check boxes are cleared</p>
<p>Send test results report when test run ends</p>	<p>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.</p> <p>Note: 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.</p> <p>To enable the notification options, you must select to Activate e-mail service option in the E-mail category.</p> <p>Default = Selected</p>

Setting E-mail Notification Options

The **E-mail** category contains options for specifying the mail server to use and other e-mail preferences.



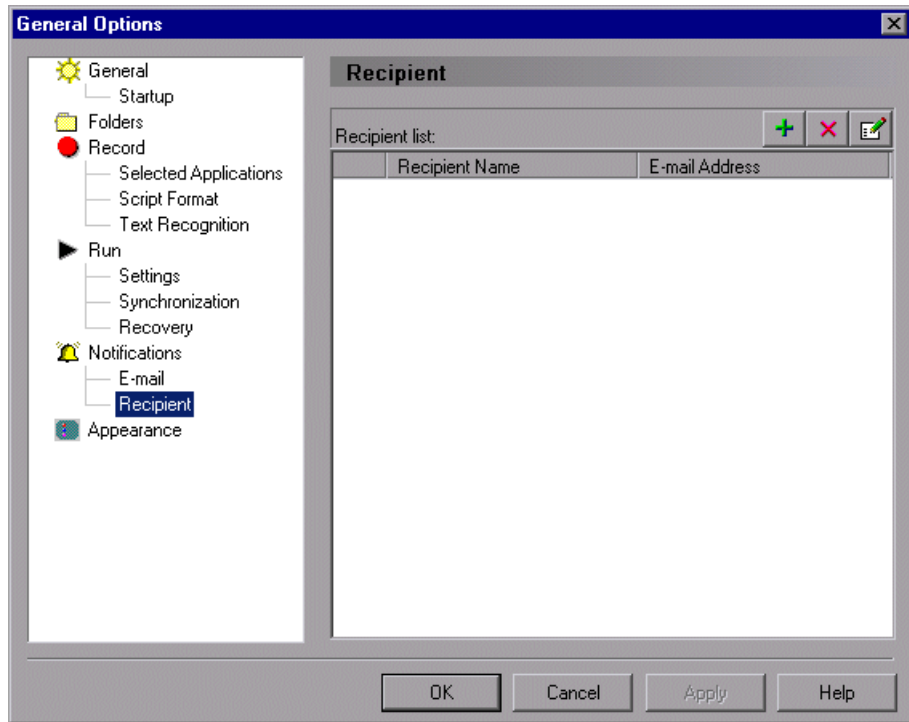
The **E-mail** category contains the following options:




Option	Description
Activate e-mail service	<p>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.</p> <p>You can use the setvar and getvar functions to set and retrieve the value of the corresponding email_service testing option from within a test script, as described in Chapter 21, “Setting Testing Options from a Test Script” in the <i>Mercury WinRunner Advanced Features User’s Guide</i>.</p> <p>Default = cleared</p>
Server address	<p>The address of the outgoing mail server you want to use to send the e-mail message.</p>
Port	<p>The mail server port to use.</p> <p>Default = 25</p>
Sender address	<p>The e-mail address you want to display as the sender of the e-mail notification. Choose one of the following:</p> <ul style="list-style-type: none"> • <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. <p>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.</p> <p>Default = <User Name>@<Host Name></p>

Option	Description
Use authentication	<p>Indicates that your outgoing mail server requires you to log in to send e-mail. When this option is selected, you must enter the login user name and password.</p> <p>Default = cleared</p>
Maximum e-mail notifications per test run	<p>The maximum number of e-mail notifications you want to send to the recipients (as specified in the Recipients category) during a test run.</p> <p>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 <i>TSL Reference</i>.</p> <p>Default = 25</p>

Setting Notification Recipients Options

The **Recipients** category enables you to specify the recipients that you want to receive e-mail notifications (according to the options selected in the **Notifications** category).

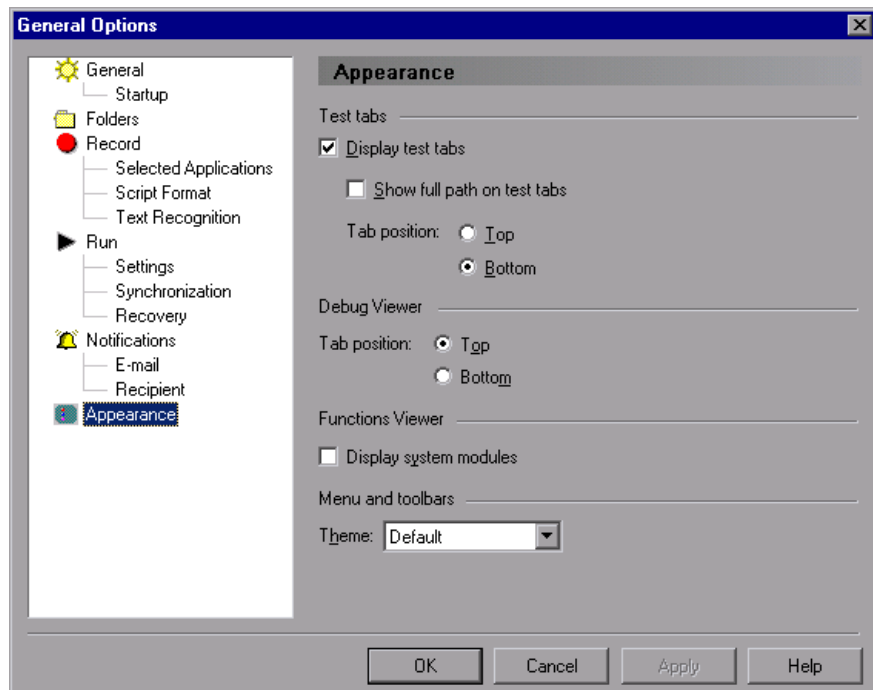


- Click **Add Recipient**  to add a new recipient to the list.
- Select a recipient from the list and click **Remove Recipient**  to remove the recipient from the list.
- Select a recipient from the list and click **Modify Recipient Details**  to modify the details of a recipient in the list.

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.



The **Appearance** category contains the following options:

Option	Description
Display test tabs	<p>Displays a tab for each open test so that you can display an open test by clicking its tab.</p> <p>If this option is cleared, you can select a test to display using the Window menu commands.</p> <p>Default = selected</p>
Show full path on test tabs	<p>When this option is selected, the full path of the test is displayed on each test tab. When this option is cleared, only the test name is displayed on the tab.</p> <p>Default = cleared</p>
Tab position (Test tabs)	<p>Indicates whether to display the test tabs at the Top or Bottom of the page.</p>
Tab position (Debug Viewer)	<p>Indicates whether to display the debug tabs at the Top or Bottom of the Debug Viewer pane.</p>
Display System Modules (Function Viewer)	<p>When this option is selected, loaded system modules are displayed in the Function Viewer.</p>
Theme	<p>Enables you to select a pre-configured style or a background image for your frame. For more information, see “Selecting a Theme” below.</p>

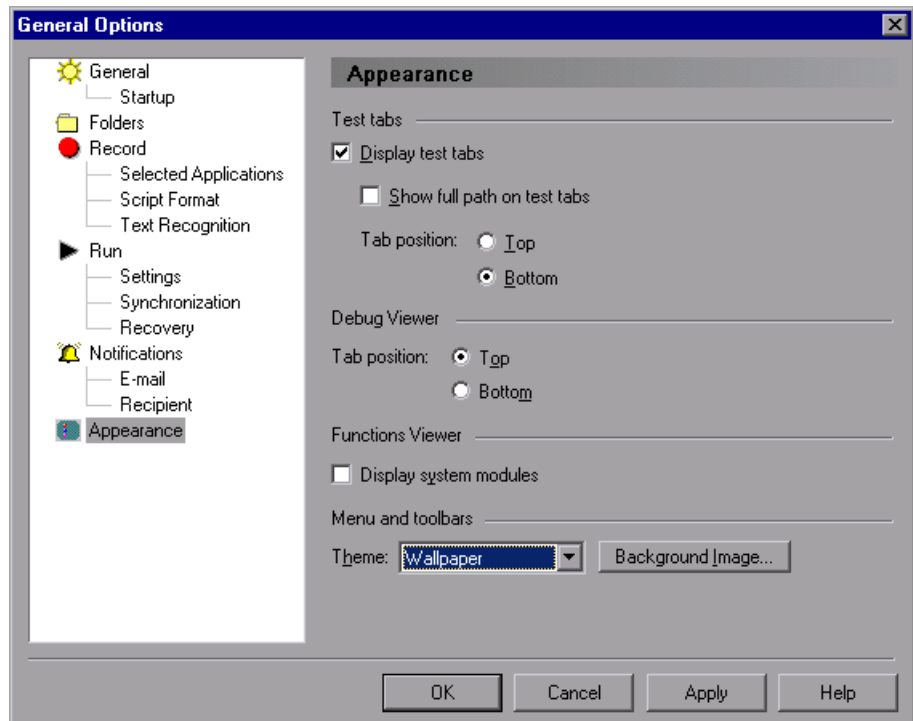
Selecting a Theme

You can select a pre-configured style for your frame from the **Theme** list. Alternatively, you can select a custom wallpaper as a background for your frame. The theme you select is reflected in both the WinRunner window and the WinRunner Test Results window.

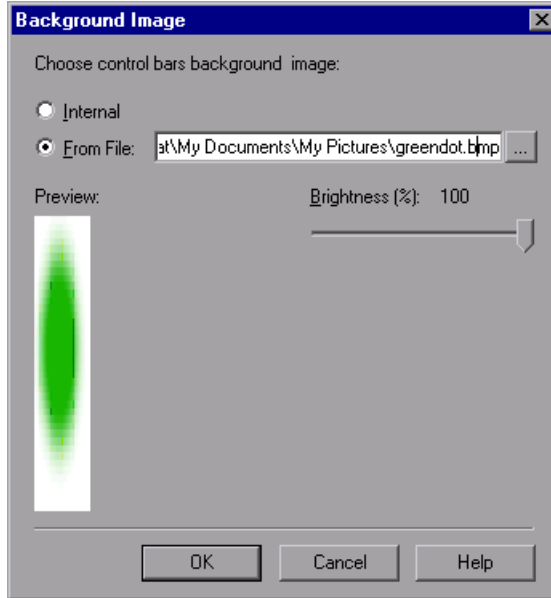
Note: The theme does not appear in the test results in the Unified report view. For more information on selecting the report view, see “Setting Test Run Options,” on page 562. For more information on the Unified report view, see “Understanding the Unified Report View Results Window,” on page 456.

To select a custom wallpaper background:

- 1 From the **Theme** list, select **Wallpaper**. The **Background Image** button is displayed.



- 2 Click **Background Image**. The Background Image dialog box opens.



- 3 Select **Internal** to use the default WinRunner background image. Select **From File** to use a custom image.
- 4 If you chose **From File** in step 3, enter a file name or use the browse button to select a bitmap (.bmp) file.
- 5 If you want to adjust the brightness of the image you selected, use the **Brightness** slider control.
- 6 Click **OK** to close the Background Image dialog box. Note that your background image appears in the WinRunner user interface only after you click **Apply** or **OK** in the General Options dialog box.

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 for Window Synchronization to maintain a constant ratio. To avoid overloading your system, you should not exceed a timeout:delay ratio of 50:1.	1000 (ms)

Setting	Description	Adjustment Recommendations	Default
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)
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)

Setting	Description	Adjustment Recommendations	Default
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
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

Setting	Description	Adjustment Recommendations	Default
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. You may also want to consider using an alternative method of checking text that does not use text recognition at all. For more information, see “Considerations for Using Text Recognition for Windows-Based Applications” on page 560.	500 (ms)

Index

Symbols

\$ symbol in Range property check 167
\
character in regular expressions 167
_web_set_tag_attr function 180

A

Acrobat Reader xvii

activating an ActiveX control 229

ActiveX controls

activating 229

checking sub-object properties
232–234

overview 218–221

retrieving properties 226–229

setting properties 226–229

support for 217–235

viewing properties 223–226

working with TSL table functions 235

ActiveX Properties Viewer. *See* GUI Spy

ActiveX tab

ActiveX, pointer values 39

ActiveX_activate_method function 229

ActiveX_get_info function 39, 226

ActiveX_set_info function 228

Add All button

in the Check GUI dialog box 151

in the Create GUI Checkpoint dialog
box 153

in the Edit GUI Checklist dialog box
156

Add button

in the Create GUI Checkpoint dialog
box 153

in the Edit GUI Checklist dialog box
156

Add Defect dialog box 504

reporting defects 504

setup 504

Add dialog box (GUI Map Editor) 86

Add-In Manager dialog box 20

add-ins 519

loading while starting WinRunner
20–22

QuickTest. *See* the WinRunner

Advanced Features User's Guide

Add-ins tab, Test Properties dialog box 519

Advanced Features User's Guide, WinRunner
xvi

Advanced Settings dialog box 359

aging

definition 444

overriding 358–363

alignment, setting 444

Analog mode 5, 103

tests recorded in XRunner 103

appearance options 585

application being tested, illustration 28

applications, startup 523–529

Argument Specification dialog box 168

arguments, specifying 164–169

DateFormat property check 166

for Compare property check 165

from the Argument Specification
dialog box 168

Range property check 167

RegularExpression property check 167

TimeFormat property check 168

associating add-ins with a test 519

Attribute/ Notation 180

AutoFill List command, data table 388

B

Background Image dialog box 588
 batch tests. *See* the WinRunner Advanced Features User's Guide
 Bitmap Checkpoint > For Object/Window 326
 Bitmap Checkpoint > For Screen Area 328
 bitmap checkpoint commands 326–329
 Bitmap Checkpoint for Object/Window button 19, 326
 Bitmap Checkpoint for Screen Area button 19, 328
 bitmap checkpoints 321–329
 Context Sensitive 326–327
 in data-driven tests 323, 397–402
 of an area of the screen 328–329
 of windows and objects 326–327
 options for failed checkpoints 324
 overview 321–323
 test results 473
 viewing results 493
 bitmap synchronization points
 in data-driven tests 423
 of objects and windows 419–420
 of screen areas 421–423
 bitmap verification. *See* bitmap checkpoints
 bitmaps
 capturing during the test run 324
 breakpoints. *See* the WinRunner Advanced Features User's Guide
 bugs. *See* defects
 button_check_info function 130, 368
 button_check_state function 368
 button_wait_info function 415

C

calendar class 158
 called tests
 Run tab settings 523
 calling functions from external libraries. *See* the WinRunner Advanced Features User's Guide
 calling tests. *See* the WinRunner Advanced Features User's Guide

Cannot Capture message
 in Database Checkpoint dialog boxes 283
 in GUI Checkpoint dialog boxes 149
 Case Insensitive Ignore Spaces verification
 databases 295
 tables 210, 257
 Case Insensitive verification
 databases 295
 tables 210, 257
 Case Sensitive Ignore Spaces verification
 databases 295
 tables 210, 257
 Case Sensitive verification
 databases 295
 tables 210, 257
 changes in GUI discovered during test run.
 See Run wizard
 Check Arguments dialog box
 for DateFormat Property check 166
 for Range property check 167
 for Regular Expression property check 167
 for TimeFormat property check 168
 CHECK BITMAP OF OBJECT/WINDOW softkey 112, 323, 326
 CHECK BITMAP OF SCREEN AREA softkey 112, 323, 328
 CHECK DATABASE (CUSTOM) softkey 112, 280
 CHECK DATABASE (DEFAULT) softkey 112, 277, 279
 Check Database dialog box 281
 Cannot Capture message 283
 Complex Value message 283
 Check Date Results dialog box 503
 Check GUI dialog box 150–152
 Cannot Capture message 149
 closing without specifying arguments 168
 Complex Value message 149
 for checking date operations 356
 for checking tables 250
 N/A message 149
 No properties are available for this object message 149

- CHECK GUI FOR MULTIPLE OBJECTS softkey
 - 112, 135
- CHECK GUI FOR OBJECT/WINDOW softkey
 - 112, 132, 133, 138, 139, 238, 239, 240
- CHECK GUI FOR SINGLE PROPERTY softkey
 - 112, 131
- Check Property dialog box 131
- check_button class 159
- check_window function 323
- checking
 - all GUI objects in a window 137–139
 - all GUI objects in a window using default checks 138
 - all GUI objects in a window while specifying checks 139
 - dates 349–358
 - multiple GUI objects in a window 135–137
 - single GUI object 132–135
 - single GUI object using default checks 132–133
 - single GUI object while specifying checks 133–135
- checking databases 259
 - overview 260–262
 - See also* databases *and* database checkpoints
- checking dates
 - in edit boxes 355–357
 - in static text fields 355–357
 - in table contents 355–357
- checking tables 247–258
 - overview 247
 - See also* tables
- checklists
 - See also* GUI checklists *or* database checklists
- checkpoints
 - bitmap 107, 321–329
 - database 259
 - GUI 107, 127–174
 - options for failed checkpoints
 - bitmap 324
 - database 262
 - GUI 129
- checkpoints (*continued*)
 - overview 107
 - text 107, 331–348
 - updating expected results 499
- Classes of Objects dialog box 151, 153, 154, 156
- Clear All button
 - in the Check GUI dialog box 151
 - in the Create GUI Checkpoint dialog box 154
 - in the Edit GUI Checklist dialog box 156
- Clear All command, data table 387
- Clear Contents command, data table 387
- Clear Formats command, data table 387
- clearing a GUI map file 87
- click_on_text functions 337, 341
- Close All command 125
- Close command 125
 - for data table 386
- closing the GUI Checkpoint dialog boxes 168
- Collapse Tree command (GUI Map Editor) 74
- column names for data tables 390
- columns, computed 245
- command line
 - running applications from 524
 - running tests from the. *See* the WinRunner Advanced Features User's Guide
- Comment command 115
- comments
 - adding to physical description 79
- Compare Expected and Actual Values button
 - in the Database Checkpoint Results dialog box 495
 - in the GUI Checkpoint Results dialog box 485
- Compare property check, specifying arguments 165
- compare_text function 342
- comparing files
 - test results 473
 - viewing results 500
- compiled modules
 - in startup functions 529

- compiled modules. *See* the WinRunner Advanced Features User's Guide
 - Complex Value message
 - in Database Checkpoint dialog boxes 283
 - in GUI Checkpoint dialog boxes 149
 - computed columns 245
 - configurations, initializing. *See* the WinRunner Advanced Features User's Guide
 - Content property check on databases 280–282
 - Context Sensitive
 - mode 4, 25, 97–100
 - recording, common problems 101–102
 - running tests, common problems 449–451
 - testing, introduction to 25–31
 - conversion file for a database checkpoint, working with Data Junction 315–316
 - Copy command 114
 - for data table 387
 - copying descriptions of GUI objects from one GUI map file to another 82
 - Create GUI Checkpoint dialog box 152–154
 - Cannot Capture message 149
 - closing without specifying arguments 168
 - Complex Value message 149
 - N/A message 149
 - No properties are available for this object message 149
 - CRV icon 11
 - currency symbols, in Range property check 167
 - Currency(0) command, data table 389
 - Currency(2) command, data table 389
 - Current Folder box 522
 - Current Line box 521
 - current test settings 521–522
 - Current Test tab, Test Properties dialog box 521
 - custom checks on databases 280–282
 - custom classes 101
 - custom execution functions 101
 - Custom Number command, data table 389
 - custom objects 101
 - custom record functions 101
 - customizing
 - WinRunner's user interface. *See* the WinRunner Advanced Features User's Guide
 - customizing test scripts. *See* the WinRunner Advanced Features User's Guide
 - customizing the Function Generator. *See* the WinRunner Advanced Features User's Guide
 - Cut command 114
 - for data table 387
 - cut-year points 352, 446
- D**
- Data Bound Grid Control 235
 - Data Comparison Viewer 487
 - Data Junction
 - choosing a database for a database checkpoint 315–316
 - default database check 279
 - TransliterationIn property 316
 - TransliterationOut property 316
 - Data menu commands, data table 388
 - data table
 - column definition 383
 - Data menu commands 388
 - declaration in manually created data-driven tests 378
 - default 396
 - Edit menu commands 387
 - editing 384–389
 - File menu commands 386
 - Format menu commands 389
 - largest number 390
 - main 396
 - maximum column width 390
 - maximum formula length 390
 - maximum number of columns 390
 - maximum number of rows 390
 - maximum row height 390
 - number precision 390

- data table (*continued*)
 - preventing data from being reformatted 385
 - row definition 383
 - saving to a new location 380
 - saving with a new name 380
 - smallest number 390
 - table format 390
 - technical specifications 390
 - valid column names 390
 - working with Microsoft Excel 386, 403
 - working with more than one data table in a test script 380
- Data Table command 385
- database checklists
 - editing 299–302
 - modifying an existing query 302–307
 - sharing 297–299
- Database Checkpoint > Custom Check
 - command, ODBC or Microsoft Query 280
- Database Checkpoint > Default Check
 - command
 - for working with Data Junction 279
 - for working with ODBC or Microsoft Query 277
- Database Checkpoint > Runtime Record
 - Check command 264
- Database Checkpoint Results dialog box
 - Cannot Capture message 283
 - Complex Value message 283
 - options 495
- Database Checkpoint wizard 283–290
 - Data Junction screens 288–290
 - ODBC/Microsoft Query screens 284–288
 - selecting a Data Junction conversion file 290
 - selecting a source query file 286
 - setting Data Junction options 289
 - setting ODBC (Microsoft Query) options 284
 - specifying an SQL statement 287
- database checkpoints
 - Database Checkpoint wizard 283–290
 - editing database checklists 299–302
 - modifying 297–307
 - modifying expected results 308–309
 - options for failed checkpoints 262
 - parameterizing 310–314
 - parameterizing queries 310
 - parameterizing SQL statements 310
 - parameterizing, guidelines 313
 - saving a database checklist to a shared folder 297–299
 - See also* runtime record checkpoints
 - test results 494
 - viewing expected results of a contents check 496
- databases
 - Case Insensitive Ignore Spaces verification 295
 - Case Insensitive verification 295
 - Case Sensitive Ignore Spaces verification 295
 - Case Sensitive verification 295
 - checking 259
 - choosing 314–316
 - connecting 317
 - creating a query in Data Junction 315–316
 - creating a query in ODBC/Microsoft Query 314–315
 - custom checks 280–282
 - Database Checkpoint wizard 283–290
 - default check with Data Junction 279
 - default check with ODBC/Microsoft Query 277–278
 - default checks 277–279
 - disconnecting 319
 - editing the expected data 296
 - importing data for data-driven tests 384
 - modifying an existing query 302–307
 - modifying checkpoints 297–307
 - Numeric Content verification 295
 - Numeric Range verification 295

- databases (*continued*)
 - overview 260–262
 - result set 260
 - retrieving information 318
 - returning the content and number of column headers 318
 - returning the last error message of the last operation for Data Junction 320
 - returning the last error message of the last operation for ODBC 319
 - returning the row content 318
 - returning the value of a single field 318
 - running a Data Junction export file 320
 - runtime record checklists, editing 272–276
 - runtime record checkpoints 264–271
 - specifying which cells to check 292
 - TSL functions for working with 316–320
 - verification method for contents of a multiple-column database 293
 - verification method for contents of a single-column database 295
 - verification type 295
 - writing the record set into a text file 319
- data-driven tests 365–410
 - analyzing test results 395
 - bitmap checkpoints 397–402
 - bitmap synchronization points 397–402
 - converting a test script manually 380–382
 - converting tests to 369–382
 - converting tests using the DataDriver wizard 369–376
 - creating a data table manually 380–382
 - creating, manually 378–382
 - DataDriver wizard 369–378
 - ddt_func.ini file 374
 - editing the data table 384–389
 - GUI checkpoints 397–402
 - data-driven tests (*continued*)
 - guidelines 409–410
 - importing data from a database 384
 - overview 366
 - process 366–396
 - running 395
 - technical specifications for the data table 390
 - using TSL functions with 402–408
 - with user-defined functions 374
 - DataDriver wizard 369–378
 - DataWindows
 - checking properties 241–243
 - checking properties of objects within 243–245
 - checking properties while specifying checks 242
 - checking properties with default checks 241
 - computed columns 245
 - Date (MM/dd/yyyy) command, data table 389
 - date field expansion 352
 - date formats
 - date operations run mode 442
 - overriding 359
 - setting 352
 - date formats supported by DateFormat
 - property check 166
 - Date Operation Run Mode dialog box 442
 - date operations run mode
 - date format 442
 - setting 442
 - date_age_string function 363
 - date_align_day function 363, 445
 - date_calc_days_in_field function 357
 - date_calc_days_in_string function 357
 - date_change_field_aging function 363
 - date_change_original_new_formats function 363
 - date_disable_format function 446
 - date_enable_format function 446
 - date_field_to_Julian function 357
 - date_is_field function 357
 - date_is_leap_year function 357
 - date_is_string function 357

- date_leading_zero function 446
- date_month_language function 358
- date_set_aging function 363, 446
- date_set_run_mode function 446
- date_set_system_date function 363
- date_set_year_limits function 446
- date_set_year_threshold function 446
- date_string_to_Julian function 358
- date_type_mode function 363

- DateFormat property check
 - available date formats 166
 - specifying arguments 166
- db_check function 261, 311
- db_connect function 317
- db_disconnect function 319
- db_dj_convert function 320
- db_execute_query function 318
- db_get_field_value function 318
- db_get_headers function 318
- db_get_last_error function 319, 320
- db_get_row function 318
- db_record_check function 264
- db_write_records function 319
- ddt_close function 372, 403
- ddt_export function 404
- ddt_func.ini file 374
- ddt_get_current_row function 406
- ddt_get_parameters function 406
- ddt_get_row_count function 372, 379, 404
- ddt_is_parameter function 406
- ddt_next_row function 404
- ddt_open function 372, 379, 386, 403
- ddt_report_row function 396, 407
- ddt_save function 372, 380, 384, 403, 410
- ddt_set_row function 379, 405
- ddt_set_val function 405, 410
- ddt_set_val_by_row function 405, 410
- ddt_show function 404
- ddt_update_from_db function 372, 380, 408
- ddt_val function 373, 382, 407
- ddt_val_by_row function 407
- Debug mode 428, 429, 437
- Debug results 429, 437
- Debug toolbar 18
- Debug Viewer pane 14

- debugging test scripts. *See* the WinRunner Advanced Features User's Guide
- declare transaction 108
- Decrease Indent command 115
- default checks
 - on a single GUI object 132–133
 - on all objects in a window 138
 - on databases 277–279
 - on standard objects 158–164

- default database check
 - with Data Junction 279
 - with ODBC/Microsoft Query 277–278
- Default Database Checkpoint button 19, 277, 279
- defects
 - reporting during a test run 506
 - reporting from Test Results window 503
- defining parameters 515
- Delete button
 - in the Create GUI Checkpoint dialog box 153
 - in the Edit GUI Checklist dialog box 156
- Delete command 114
 - for data table 387
- deleting objects from a GUI map file 86
- Description tab, Test Properties dialog box 514
- descriptions. *See* physical descriptions
- descriptive test information 514
- dialog boxes for interactive input
 - creating. *See* the WinRunner Advanced Features User's Guide
- Display button, in Test Results window 499
- documentation
 - updates xviii
- documentation, printed
 - WinRunner Advanced Features User's Guide xvi
- DropDown DataWindows. *See* DropDown objects
- DropDown lists. *See* DropDown objects

DropDown objects
 checking properties including contents 238–241
 checking properties while specifying checks 239
 checking properties with default checks 239
drop-down toolbar, recording on a 106
DropDownListBoxContent property check 239
DWComputedContent property check 245
DWTableContent property check 241

E

Edit Check dialog box 205
 editing the expected data 211, 258, 296
 for a multiple-column database 291
 for a multiple-column table 253
 for a single-column database 294
 for a single-column table 209, 256
 for checking databases 291–297
 for checking tables 253–258
 specifying which cells to check 206, 254, 292
 verification method 207, 255, 293
 verification type 210, 257, 295
edit class 159
Edit Database Checklist command 298, 299
Edit Database Checklist dialog box 300, 303, 306
 Modify button 304, 307
Edit Expected Value button 170–171
Edit GUI Checklist command 144, 145, 155
Edit GUI Checklist dialog box 155–157
 closing without specifying arguments 168
 No properties are available for this object message 149
Edit menu commands, data table 387
Edit Runtime Record Checklist command 272
edit_check_info function 130, 368
edit_check_selection function 368
edit_wait_info function 415

editing
 database checklists 299–302
 expected property values 170–171
 GUI checklists 144–148
 GUI map 89
 runtime record checklists 272–276
 tests 114
end transaction 108
error handling. *See* the WinRunner Advanced Features User's Guide
Excel. *See* Microsoft Excel
execution arrow 16, 95
Expand Tree command (GUI Map Editor) 74
Expected Data Viewer 492, 498
expected results 431, 438, 439
 creating multiple sets 439
 specifying 440
 updating 431
 updating for bitmap, GUI, and database checkpoints 499
Expected Results Folder box 522
expected results folder, location 522
expected results of a GUI checkpoint 140
 editing 170–171
 modifying 172–174
Export command, data table 386
exporting tests to zipped files 124
extracting WinRunner tests 124

F

FarPoint Spreadsheet Control 235
file management 116
File menu commands, data table 386
File toolbar 17
file_compare function 500
Fill Down command, data table 387
Fill Right command, data table 387
filtering results, unified report 462
Filters dialog box (GUI Map Editor) 88
filters in GUI Map Editor 88
Find command 115
 for data table 387
Find in GUI Map command 41
Find Next command 115
Find Previous command 115

find_text function 337–339
 Fixed command, data table 389
 floating toolbar 17
 folder options 541
 font group

- creating 345–346
- definition 343
- designating the active 347

 Font Groups dialog box 345
 font library 343
 fonts

- learning 343–344
- teaching to WinRunner 342–348

 Fonts Expert 343
 Format menu commands, data table 389
 Fraction command, data table 389
 frame object properties 181
 Function Generator. *See the WinRunner Advanced Features User's Guide*
 functions

- calling from external libraries. *See the WinRunner Advanced Features User's Guide*
- defining startup 527
- startup 523–529
- user-defined. *See the WinRunner Advanced Features User's Guide*

G

General command, data table 389
 General Options

- Appearance 585
- Folder 541
- General 535
- Startup 538
- Notification 579
- Record 545

 General Options dialog box 522, 531
 General tab

- Test Properties dialog box 396, 512

 generating functions. *See the WinRunner Advanced Features User's Guide*
 generic object class 162
 Get Text > From Object/Window command 334

Get Text from Object/Window button 19, 334
 GET TEXT FROM OBJECT/WINDOW softkey 113, 334
 Get Text from Screen Area button 19, 335
 Get Text from Screen Area command 335
 GET TEXT FROM SCREEN AREA softkey 113, 335
 get_text function 333–336
 Global GUI Map File mode 45–64, 537

- guidelines 63–64
- overview 45–47

 global testing options. *See setting global testing options*
 Go To command, for data table 388
 GUI

- changes discovered during test run. *See Run wizard*
- learning 40, 48–54
- teaching to WinRunner 40, 48–54

 GUI checklists 140

- editing 144–148
- modifying 143–148
- sharing 143–144
- using an existing 141–143

 GUI Checkpoint > For Single Property command

- with data-driven tests 367

 GUI Checkpoint commands 132, 133, 135, 138, 139
 GUI Checkpoint dialog boxes 148–157
 GUI Checkpoint for Multiple Objects button 19, 135, 142, 152

- See also* GUI Checkpoint for Multiple Objects command

 GUI Checkpoint for Multiple Objects command 135, 142, 152
 GUI Checkpoint for Object/Window button 19, 132, 133, 138, 139, 150

- See also* GUI Checkpoint for Object/Window command

 GUI Checkpoint for Object/Window command 132, 133, 138, 139, 150
 GUI Checkpoint for Single Property command 131

- with data-driven tests 397

- GUI Checkpoint Results dialog box 484
 - Cannot Capture message 149
 - Complex Value message 149
 - N/A message 149
 - No properties are available for this object message 149
 - options 485
 - Update Expected Value button 499
- GUI checkpoints 127–174, 175–216
 - checking a single object 132–135
 - checking a single object using default checks 132–133
 - checking a single object while specifying checks 133–135
 - checking all objects in a window 137–139
 - checking all objects in a window using default checks 138
 - checking all objects in a window while specifying checks 139
 - checking multiple objects in a window 135–137
 - checking text in Web objects 212–216
 - checking Web objects 175–216
 - default checks 158–164
 - editing expected property values 170–171
 - editing GUI checklists 144–148
 - GUI Checkpoint dialog boxes 148–157
 - in data-driven tests 397–402
 - modifying expected results 172–174
 - modifying GUI checklists 143–148
 - on dates 355–357
 - options for failed checkpoints 129
 - overview 128–129
 - property checks 158–164
 - saving a GUI checklist to a shared folder 143–144
 - specifying arguments 164–169
 - test results 473, 484
 - using an existing GUI checklist 141–143
- GUI checkpoints on dates 355–357
 - test results 502
- GUI checks
 - on standard objects 158–164
 - specifying arguments for 164–169
- GUI Files command (GUI Map Editor) 74
- GUI map
 - configuring. *See* the WinRunner Advanced Features User's Guide
 - creating 48–56
 - finding objects or windows 41
 - introduction 25–31
 - loading 59
 - overview 33–34, 47–48
 - saving 57–59
 - understanding 33–43
 - viewing 30
- GUI Map command (GUI Map Editor) 74
- GUI Map Configuration
 - Web objects 177
- GUI Map Editor 73–79
 - copying/moving objects between files 82
 - deleting objects 86
 - description of 74
 - expanded view 83
 - filtering displayed objects 88
 - introduction 30
 - learning the GUI of an application 48–56
 - loading GUI files 61
- GUI Map File modes
 - comparison of 42–43
 - Global GUI Map File 537
 - Global GUI Map File mode 45–64
 - GUI Map File per Test 537
 - GUI Map File per Test mode 65–69
- GUI Map File per Test mode 65–69, 537
 - guidelines 69
 - overview 65–66
 - setting option 67–68
 - updating a GUI map file 68
- GUI map files
 - adding objects 86
 - clearing 87
 - copying/moving objects between files 82
 - deleting objects 86

GUI map files (*continued*)

- editing 89
- finding a single object 84
- finding multiple objects 85
- guidelines 41
- loading 59
- loading using the GUI Map Editor 61
- loading using the GUI_load function 59
- merging. *See* the WinRunner
 - Advanced Features User's Guide
- saving 57–59
- saving changes 89
- saving temporary 57
- sharing among tests 47–48
- tracing objects between files 85
- updating in GUI Map File per Test mode 68

GUI object properties, viewing 34–39

GUI objects

- checking 127–174
- checking property values 130–132
- identifying 25–31

GUI Spy 34–39

- ActiveX tab 38, 223–226
- All standard tab 35
- Recorded tab 36

GUI Test Builder. *See* GUI Map Editor

GUI_close function 60

GUI_load function 59, 450

GUI_open function 60

GUI_unload function 60

GUI_unload_all function 60

gui_ver_add_class function 151, 154, 156

gui_ver_set_default_checks function 132, 137

H

html_check_button object 177

html_combobox object 177

html_edit object 177

html_frame object 177

html_listbox object 177

html_push_button object 177

html_radio_button object 177

html_rect object 177

html_text_link object 177

HWND window handle 177

I

Import command, data table 386

importing data from a database, for a data-driven test 384

- Microsoft Query file, existing 393
- Microsoft Query file, new 392
- Microsoft Query options 391
- specifying SQL statement 394
- using Microsoft Query 391–395

importing tests to zipped files 124

Increase Indent command 115

incremental aging 442

initialization tests. *See* the WinRunner

- Advanced Features User's Guide

input parameters 447, 515

Insert command, data table 387

Insert Function for Object/Window button 19

INSERT FUNCTION FOR OBJECT/WINDOW softkey 113

Insert Function from Function Generator button 19

INSERT FUNCTION FROM FUNCTION GENERATOR softkey 112, 113

invoke_application function 105, 524

K

key assignments

- default 111, 435

keyboard shortcuts 111, 435

L

labels, varying 79

Learn Font dialog box 344

learning the GUI of an application 48–56

- by recording 54–55
- with the GUI Map Editor 55–56
- with the RapidTest Script wizard 49–54

Index

- learning the GUI of your application 40, 49–54
- IFPSpread.Spread.1 MSW_class. *See* FarPoint Spreadsheet Control
- list class 161
- list_check_info function 130, 368
- list_check_item function 368
- list_check_selected function 368
- list_wait_info function 415
- load function 434
- load testing. *See* the WinRunner Advanced Features User's Guide
- loading add-ins 519
 - while starting WinRunner 20–22
- loading the GUI map file 59
 - using the GUI Map Editor 61
 - using the GUI_load function 59
- loading WinRunner add-ins 20–22
- LoadRunner
 - description 10
- LoadRunner. *See* the WinRunner Advanced Features User's Guide
- location
 - current working folder 522
 - expected results folder 522
 - verification results folder 522
- logical name
 - definition 29
 - for Web objects, setting properties for 180
 - modifying 43, 63, 77–79

M

- main data table 396
- managing the testing process. *See* the WinRunner Advanced Features User's Guide
- matching database fields
 - when creating runtime record checkpoints 267
 - when editing runtime record checklists 274
- menu bar, WinRunner 14
- menu_item class 161
- menu_select_item function 106

- menu_wait_info function 415
- menu-like toolbar, recording on 106
- merging GUI map files. *See* the WinRunner Advanced Features User's Guide
- messages
 - in the Database Checkpoint dialog boxes 283
 - in the GUI Checkpoint dialog boxes 149
- Microsoft Excel, with data tables 386, 403
- Microsoft Grid Control 235
- Microsoft Query
 - and runtime record checkpoints 265
 - choosing a database for a database checkpoint 314–315
 - default database check 277–278
 - importing data from a database 391–395
- minimizing WinRunner, when recording a test 105
- Modify button, in Edit Database Checklist dialog box 304, 307
- Modify dialog box (GUI Map Editor) 78
- Modify ODBC Query dialog box 304
- modifying
 - expected results of a database checkpoint 308–309
 - expected results of a GUI checkpoint 172–174
 - GUI checklists 143–148
 - logical names of objects 43, 63, 77–79
 - physical descriptions of objects 77–79
- modules, compiled. *See* the WinRunner Advanced Features User's Guide
- move_locator_text function 339–340
- MSDBGrid.DBGrid MSW_class. *See* Data Bound Grid Control
- MSGrid.Grid MSW_class. *See* Microsoft Grid Control

N

- N/A message, GUI Checkpoint dialog boxes 149
- names. *See* logical names
- New button 116

New command 116
 for data table 386
 New icon in Runtime Record Checkpoint wizard 275
 No properties are available for this object message, in GUI Checkpoint dialog boxes 149
 No properties were captured for this object message, in GUI Checkpoint dialog boxes 149
 nonstandard properties 154, 157
 notification options 579
 Numeric Content verification
 databases 295
 tables 210, 257

Numeric Range verification
 databases 295
 tables 210, 257

O

obj_check_bitmap function 327
 in data-driven tests 397
 obj_check_gui function 140–141, 355, 357
 in data-driven tests 397
 obj_check_info function 130, 368
 obj_check_text function 337
 obj_click_on_text 340–341
 obj_exists function 414
 obj_find_text function 338–339
 obj_get_text function 333–336
 obj_mouse function 101
 obj_move_locator_text 339–340
 obj_wait_bitmap function 420
 in data-driven tests 397
 obj_wait_info function 415
 object class 162
 object synchronization points 413–414
 objects
 finding in the GUI map 41
 virtual. *See* the WinRunner Advanced Features User's Guide
 OCX controls. *See* ActiveX controls
 OCX Properties Viewer. *See* GUI Spy
 ActiveX tab

ODBC
 choosing a database for a database checkpoint 314–315
 default database check 277–278
 OLE controls. *See* ActiveX controls
 online resources xvii
 Open button
 in the Create GUI Checkpoint dialog box 153
 in the Edit GUI Checklist dialog box 156
 Open Checklist dialog box
 for database checklists 298, 300, 303, 306
 for GUI checklists 142, 145
 Open command
 for data table 386
 Open GUI File dialog box 61
 Open GUI File from Quality Center Project dialog box 62
 Open or Create a Data Table dialog box 377, 384, 385
 Open Test dialog box 120
 Open Test from Quality Center Project dialog box 121
 opening test results, unified report 477
 opening tests 116
 from file system 120
 options, global testing. *See* setting global testing options
 output parameters 515
 Override Aging dialog box 360, 362
 Override Object Settings dialog box 361
 overriding
 date formats 359
 date objects 360
 date settings 358–363

P

Parameterize Data command 381
 Parameterize Data dialog box 381
 parameterizing database checkpoints
 310–314
 guidelines 313

Index

- parameterizing database checkpoints
 - (*continued*)
 - SQL statements 310
- parameters
 - defining for a test 515, 517
 - input 515
 - managing for a test 515
 - output 515
- Parameters tab, Test Properties dialog box 515
- Paste command 114
 - for data table 387
- Paste Values command, data table 387
- Pause button 435
- Pause command 435
- PAUSE softkey 436

- pausing test execution using breakpoints. *See* the WinRunner Advanced Features User's Guide
- Percent command, data table 389
- physical descriptions
 - adding comments to 79
 - changing regular expressions in 81
 - definition 27–29
 - modifying 77–79
- pointer values, ActiveX 39
- PowerBuilder
 - DataWindows 241–243, 243–245, 245
 - DropDown objects 238–241
- PowerBuilder applications 237–245
 - overview 238
- previewing test results 464
- Print command 125
 - for data table 386
- Print Setup command, data table 386
- printing, test results 463
- problems
 - recording Context Sensitive tests 101–102
 - running Context Sensitive tests 449–451
- programming in TSL. *See* the WinRunner Advanced Features User's Guide
- properties
 - setting test 510
 - test 509–529
- properties of ActiveX controls
 - retrieving 226–229
 - setting 226–229
 - viewing 223–226
- properties of Visual Basic controls
 - retrieving 226–229
 - setting 226–229
 - viewing 223–226
- property checks
 - checking property values 130–132
 - on standard objects 158–164
 - specifying arguments 164–169
 - test results 483
- Property List button 151, 154, 156

- property value synchronization points 414–418
- property values, editing 170–171
- push_button class
 - push button objects 162

- Q**
- qcdb_add_defect function 503, 506
- Quality Center 520
 - Add Defect dialog box 504
 - connecting from unified report 479
 - description 9
 - reporting defects during a test run 506
 - TdApiWnd icon 11
 - working with
- Quality Center Connection dialog box 479
- query file for a database checkpoint, working with ODBC/Microsoft Query 314–315
- QuickTest
 - loading associated add-ins
 - supported versions
- quotation marks in GUI map files 75, 79

R

- radio_button class 159
- Range property check
 - currency symbols 167
 - specifying arguments 167
- RapidTest Script wizard
 - learning the GUI of an application 49–54
- Read Me file xvii
- reading text 333–336
 - from an area of an object or a window 335
 - in a window or an object 334
- Recalc command, data table 388
- Record - Analog command 100
- Record - Context Sensitive button 18, 19
- Record - Context Sensitive command 100
- Record button 100
- Record commands 100
- RECORD softkey 112
- Record/Run Engine icon 11
- recording
 - options 545
 - problems while 101–102
- recording tests
 - Analog mode 103
 - Context Sensitive mode 97–100
 - guidelines 97
 - with WinRunner minimized 105
- recovery scenarios. *See* the WinRunner Advanced Features User's Guide
- regular expressions
 - changing, in the physical description 81
 - character 167
- regular expressions. *See* the WinRunner Advanced Features User's Guide
- RegularExpression property check, specifying arguments 167
- Replace command 115
 - for data table 388
- reporting defects from Test Results window 503
- result set 260
- results display, customizing 466
- results folders
 - debug 437
 - expected 431, 439
 - verify 429, 436
- results of tests. *See* test results
- results of tests. *See* the WinRunner Basic Features User's Guide xvi
- results schema 466
- RTL-style windows
 - WinRunner support for applications with 106
- Run commands 433
- Run from Arrow
 - button 18, 433
 - command 433
- RUN FROM ARROW softkey 435
- Run from Top
 - button 18, 433
 - command 433, 523
- Run from Top command 523
- RUN FROM TOP softkey 435
- Run Minimized > From Arrow command 434
- Run Minimized > From Top command 434, 523
- Run Minimized commands 433
- Run Mode
 - box 522
- Run Mode button 18
- run modes
 - Debug 428, 429, 437
 - displaying for current test 522
 - Update 428, 431
 - Verify 428, 429
- Run tab, Test Properties dialog box 523
- Run Test dialog box 429, 437, 441
 - for date operations 445
- Run wizard 75–77
- running tests 427–448
 - batch run. *See* the WinRunner Advanced Features User's Guide
 - checking your application 436
 - controlling with configuration parameters 448
 - controlling with test options 448
 - debugging a test script 437

- running tests (*continued*)
 - for debugging. *See* the WinRunner Advanced Features User's Guide
 - from the command line. *See* the WinRunner Advanced Features User's Guide
 - overview 428
 - problems while 449–451
 - run modes 428
 - setting global testing options 562–578
 - to check date operations 442–446
 - updating expected results 438
 - runtime database record checklists, editing 272–276
 - runtime database record checkpoints 264–271
 - runtime record checklists, editing 272–276
 - Runtime Record Checkpoint wizard 264–276
 - New icon 275
 - runtime record checkpoints 264–271
 - changing success conditions 276
 - comparing data in different formats 269
 - specify number of matching database records 269
- S**
- sample tests xvii
 - Save All command 116
 - Save As button
 - in the Create GUI Checkpoint dialog box 153
 - in the Edit GUI Checklist dialog box 156
 - Save As command 116
 - for data table 386
 - Save button 116
 - Save Checklist dialog box
 - for database checklists 299
 - for GUI checklists 144
 - Save command 116
 - for data table 386
 - Save GUI File dialog box 57
 - Save GUI File to Quality Center Project dialog box 58
 - Save Test dialog box 117
 - Save Test to Quality Center Project dialog box 118
 - saving
 - GUI map files 57–59
 - temporary GUI map file 57
 - saving changes to the GUI map file 89
 - saving tests
 - in file system 116
 - in Quality Center project database 118
 - schema, for results 466
 - Scientific command, data table 389
 - Script wizard. *See* RapidTest Script wizard
 - scroll class 163
 - scroll_check_info function 130, 368
 - scroll_check_pos function 368
 - scroll_wait_info function 415
 - Section 508 94
 - Select All button
 - in the Check GUI dialog box 151
 - in the Create GUI Checkpoint dialog box 153
 - in the Edit GUI Checklist dialog box 156
 - Select All command 114
 - Set Date Formats dialog box 353
 - set_window function 31
 - setting date formats 353
 - setting global testing options 531–592
 - current test settings 522
 - running a test 562–578
 - text recognition 558–560
 - setting test properties 509–529
 - add-ins 519
 - documenting descriptive test information 514
 - documenting general test information 512
 - parameters 515
 - test properties dialog box 510
 - setting test properties. *See* the WinRunner Basic Features User's Guide xvi

- setting testing options
 - globally 531–592
 - within a test script. *See the WinRunner Advanced Features User's Guide*
- setting the date operations run mode 442
- setvar function 448
- shared folder
 - for database checklists 297–299
 - for GUI checklists 143–144
- sharing GUI map files among tests 47–48
- Sheridan Data Grid Control 235
- Show All Properties button
 - in the Check GUI dialog box 152
 - in the Create GUI Checkpoint dialog box 154
 - in the Database Checkpoint Results dialog box 496
 - in the Edit GUI Checklist dialog box 157
 - in the GUI Checkpoint Results dialog box 485
- Show Failures Only button
 - in the Database Checkpoint Results dialog box 495
 - in the GUI Checkpoint Results dialog box 485
- Show Nonstandard Properties Only button
 - in the Check GUI dialog box 152
 - in the Create GUI Checkpoint dialog box 154
 - in the Database Checkpoint Results dialog box 496
 - in the Edit GUI Checklist dialog box 157
 - in the GUI Checkpoint Results dialog box 485
- Show Selected Properties Only button
 - in the Check GUI dialog box 151
 - in the Create GUI Checkpoint dialog box 154
 - in the Edit GUI Checklist dialog box 157
- Show Standard Properties Only button
 - in the Check GUI dialog box 152
 - in the Create GUI Checkpoint dialog box 154
 - in the Database Checkpoint Results dialog box 495
 - in the Edit GUI Checklist dialog box 157
 - in the GUI Checkpoint Results dialog box 485
- Show TSL button, in the WinRunner Test Results window 173, 308
- Show User Properties Only button
 - in the Check GUI dialog box 152
 - in the Create GUI Checkpoint dialog box 154
 - in the Edit GUI Checklist dialog box 157
 - in the GUI Checkpoint Results dialog box 485
- softkeys
 - default settings 111, 435
- Sort command, data table 388
- Specify 'Compare' Arguments dialog box 165
- Specify Arguments button 164–169
- Specify Text dialog box 213, 215
- specifying arguments 164–169
 - for DateFormat property check 166
 - for Range property check 167
 - for RegularExpression property check 167
 - for TimeFormat property check 168
 - from the Argument Specification dialog box 168
- specifying which checks to perform on all objects in a window 139
- specifying which properties to check for a single object 133–135
- spin_wait_info function 415
- spying on GUI objects 34–39
- SQL statements
 - and creating runtime record checkpoints 266
 - creating result sets based on 318
 - executing queries from 318

- SQL statements (*continued*)
 - for editing runtime record checklists 273
 - parameterizing in database checkpoints 310
 - specifying in the Database Checkpoint wizard 287
 - SSDataWidgets.SSDBGridCtrl.1. *See* Sheridan Data Grid Control
 - standard objects
 - default checks 158–164
 - property checks 158–164
 - standard properties 154, 157
 - Standard toolbar 17
 - start transaction 108
 - starting WinRunner, with add-ins 20–22
 - startup applications and functions 523–529
 - startup functions 527
 - compiled modules in 529
 - startup options 538
 - static aging 442
 - static_check_info function 130, 368
 - static_check_text function 368
 - static_text class 159
 - static_wait_info function 415
 - status bar, WinRunner 14
 - statusbar_wait_info function 415
 - Step button 434
 - Step command 434
 - Step Into button 434
 - Step Into command 434
 - STEP INTO softkey 435
 - Step Out command 434
 - STEP OUT softkey 436
 - STEP softkey 435
 - Step to Cursor command 434
 - STEP TO CURSOR softkey 436
 - Stop button 18, 19, 100, 104, 434
 - Stop command 434
 - Stop Recording command 100, 104
 - STOP softkey 113, 436
 - supplying values 447
 - synchronization
 - waiting for bitmaps of objects and windows 419–420
 - synchronization (*continued*)
 - waiting for bitmaps of screen areas 421–423
 - waiting for objects 413–414
 - waiting for property values 414–418
 - waiting for windows 413–414
 - Synchronization Point for Object/Window Bitmap button 19, 419
 - Synchronization Point for Object/Window Bitmap command 419
 - Synchronization Point for Object/Window Property button 19, 416
 - Synchronization Point for Object/Window Property command 416
 - Synchronization Point for Screen Area Bitmap button 19, 422
 - Synchronization Point for Screen Area Bitmap command 422
 - synchronization points 108
 - in data-driven tests 397–402
 - SYNCHRONIZE BITMAP OF OBJECT/WINDOW softkey 112, 419, 423
 - SYNCHRONIZE BITMAP OF SCREEN AREA softkey 112, 422, 423
 - SYNCHRONIZE OBJECT PROPERTY (CUSTOM) softkey 112
 - synchronizing tests 411–423
 - tips 423
 - system variables. *See* setting testing options
- T**
- tab_wait_info function 415
 - TableContent property check 250–252
 - tables
 - Case Insensitive Ignore Spaces verification 210, 257
 - Case Insensitive verification 210, 257
 - Case Sensitive Ignore Spaces verification 210, 257
 - Case Sensitive verification 210, 257
 - checking 247–258
 - checking contents while specifying checks 250–252
 - checking contents with default checks 249

- tables (*continued*)
 - editing the expected data 211, 258
 - Numeric Content verification 210, 257
 - Numeric Range verification 210, 257
 - overview 247
 - specifying which cells to check 206, 254
 - verification method for contents of a single-column database 209, 256
 - verification method for multiple-column tables 207, 255
 - verification type 210, 257
 - viewing expected results of a contents check 489
 - viewing results of a contents check 486
- tbl_activate_cell function 235
- tbl_activate_header function 235
- tbl_get_cell_data function 235
- tbl_get_cols_count function 235
- tbl_get_column_name function 235
- tbl_get_rows_count function 235
- tbl_get_selected_cell function 235
- tbl_get_selected_row function 235
- tbl_select_col_header function 235
- tbl_set_cell_data function 235
- tbl_set_selected_cell function 235, 238, 239, 240
- tbl_set_selected_row function 235
- TdApiWnd icon 11
- tddb_functions. *See* qcdb_functions
- teaching WinRunner the GUI of an application 48–56
 - by recording 54–55
 - from the GUI Map Editor 55–56
 - overview 40, 47–54
 - with the RapidTest Script wizard 49–54
- technical support online xviii
- temporary GUI map file, saving 57
- test editor window 16
- test execution
 - See also* running tests
- test information 512
- test log 473
- test parameters 515
- test properties 509–529
- Test Properties dialog box
 - Add-ins 519
 - Current Test tab 521
 - Description tab 514
 - General tab 396, 512
 - Parameters tab 515
 - Run tab 523
- test properties, setting 510
- test results 453–506
 - bitmap checkpoints 473, 493
 - checkpoint results 481
 - database checkpoints 494, 496
 - file comparison 473
 - GUI checkpoints 473, 484
 - GUI checkpoints on dates 502
 - property checks 483
 - reporting defects 503
 - tables 486
 - unified report view 456
 - updating expected 499
 - viewing from a Quality Center project database 475–477
 - viewing, overview 474–477
 - WinRunner report view 467
- test results display, customizing 466
- Test Results window 467–474, 475
 - Display button 499
 - test log 473
 - test summary 471
 - test tree 471
- test run
 - viewing results 474–477
- Test Script Language (TSL). *See* the WinRunner Advanced Features User's Guide
- test scripts 16, 95
 - customizing. *See* the WinRunner Advanced Features User's Guide
- test settings
 - current 522
 - current, Test Properties dialog box Current Test tab 521
- test summary 471
- Test toolbar 14, 18

Index

- test tree 471
- test window
 - WinRunner 95
- test wizard. *See* RapidTest Script wizard
- testing options 448
 - global. *See* setting global testing options
 - options
 - within a test script. *See* the WinRunner Advanced Features User's Guide
- testing process
 - analyzing results 453–506
 - introduction 5
 - managing the
 - running tests 427–448
- tests
 - calling. *See* the WinRunner Advanced Features User's Guide
 - checkpoints 107
 - creating 93–125
 - documenting descriptive test information 514
 - documenting general test information 512
 - editing 114
 - extracting 124
 - new 116
 - opening existing 116
 - planning 111
 - previewing results 464
 - printing results 463
 - programming 106
 - recording 97–103
 - synchronization points 108
 - zipping 124
- TestSuite 9
- text
 - checking 331–348
 - comparing 342
 - location 338–339
 - reading 333–336
 - searching for 337–341
- text checkpoints 331–348
 - comparing text 342
 - creating a font group 345–346
 - overview 331–332
- text checkpoints (*continued*)
 - reading text 333–336
 - searching for text 337–341
 - teaching fonts to WinRunner 342–348
- text link properties 182
- text recognition
 - options 558–560
 - risks and alternatives 560
- text string
 - clicking a specified 340–341
 - moving the pointer to a 339–340
- themes 586
- threshold 352, 446
- time formats supported by TimeFormat
 - property check 168
- Time h mm AM/PM command, data table 389
- TimeFormat property check
 - available time formats 168
 - specifying arguments 168
- title bar, WinRunner 14
- toolbar
 - creating a floating 17
 - Debug 18
 - File 17
 - Test 14, 18
 - User 14, 19
- toolbar_select_item function 106
- transactions 108
- True DBGrid Control 235
- TrueDBGrid50.TDBGrid MSW_class. *See* True DBGrid Control
- TrueDBGrid60.TDBGrid MSW_class. *See* True DBGrid Control
- TrueOleDBGrid60.TDBGrid MSW_class. *See* True DBGrid Control
- TSL functions
 - for working with a database 316–320
 - with data-driven tests 402–408
- TSL Online Reference xvii
- TSL Reference Guide xvii
- typographical conventions xix

U

Uncomment command 115
 Undo command 114
 unified report 456

- connecting to Quality Center 479
- filtering results 462
- finding results 461
- menu bar and toolbar 458
- opening test results 477

 unified report view, definition 454
 unmapped classes. *See* object class
 unzipping WinRunner tests 124
 Update Expected Value button

- in the Database Checkpoint Results dialog box 495
- in the GUI Checkpoint Results dialog box 485, 499

 Update mode 428, 431
 updates, documentation xviii
 user interface, customizing. *See* the WinRunner Advanced Features User's Guide
 User properties 152, 154, 157, 485
 User toolbar 14, 19
 user-defined functions

- adding to the Function Generator. *See* the WinRunner Advanced Features User's Guide
- parameterizing for data-driven tests 374

 user-defined functions. *See* the WinRunner Advanced Features User's Guide
 user-defined properties 152, 154, 157, 485

V

valid column names for data tables 390
 Validation Rule command, data table 389
 variables

- monitoring. *See* the WinRunner Advanced Features User's Guide

 verification method

- for databases 293
- for tables 207, 255

 verification results 429, 436
 Verification Results Folder box 522

verification results folder, location 522
 verification type

- for databases 295
- for tables 210, 257

 verification, bitmap. *See* bitmap checkpoints
 Verify mode 428, 429, 436
 viewing test results

- previewing test results 464
- printing test results 463

 viewing, GUI object properties 34–39
 virtual objects. *See* the WinRunner Advanced Features User's Guide
 Visual Basic controls

- checking sub-object properties 232–234
- overview 218–221
- retrieving properties 226–229
- setting properties 226–229
- support for 217–235
- viewing properties 223–226

W

wait_window function 423
 wallpaper 586

- setting custom background 587

 Watch List. *See* the WinRunner Advanced Features User's Guide
 WDiff utility 500
 Web exception handling. *See* the WinRunner Advanced Features User's Guide
 Web image properties 182
 Web objects 175–216

- check box object properties 185
- checking 187–216
- checking broken links 199–202
- checking content of frames, cells, links, or images 193–194
- checking font or color of text links 198–199
- checking number of columns and rows 194–195
- checking object count in frames 190–191
- checking standard frame properties 188–190

Web objects (continued)

- checking structure of frames, tables, and cells 191–192
 - checking table content 203–204
 - checking text 212–216
 - checking the URL of links 195–196
 - edit box object properties 185
 - frame object properties 181
 - list and combo box object properties 186
 - properties for all objects 179
 - radio button properties 184
 - text link properties 182
 - using properties in your test 178–187
 - viewing recorded properties 176
 - Web button object properties 187
 - Web image properties 182
 - Web table cell properties 183
 - Web table properties 183
 - working with 175–216
- Web radio button properties 184
- Web table cell properties 183
- Web table properties 183
- web_frame_get_text 212, 213
- web_frame_text_exists 212, 214
- web_obj_get_text 212, 213
- web_obj_text_exists 212, 214
- WebTest add-in 102
- with GUI Map Configuration 177
- Welcome to WinRunner window 13
- What's New in WinRunner help xvii
- win_activate function 105
- win_check_bitmap function 327, 329
- in data-driven tests 397
- win_check_gui function 140–141
- in data-driven tests 397
- win_check_info function 130, 368
- win_check_text function 337
- win_click_on_text 340–341
- win_exists function 414
- win_find_text function 338–339
- win_get_text function 333–336
- win_move_locator_text 339–340
- win_wait_bitmap function 420
- in data-driven tests 397
- win_wait_info function 415

- window class 163
- window labels, varying 79
- window synchronization points 413–414
- windowing 352
- WinRunner 16
- introduction 3–10
 - main window 14
 - menu bar 14
 - online resources xvii
 - overview 11–22
 - starting 11–13
 - status bar 14
 - test editor window 16
 - test window 16
 - title bar 14
- WinRunner Context-Sensitive Help xvii
- WinRunner Customization Guide xvii
- WinRunner Installation Guide xvi
- WinRunner Quick Preview xvii
- WinRunner Record/Run Engine icon 11
- WinRunner report 467
- menu bar and toolbar 469
 - test log 473
 - test summary 471
- WinRunner report view
- definition 454
- WinRunner support for applications with RTL-style windows 106
- WinRunner Test Results window 467–474, 475
- for expected results of a GUI checkpoint 172
- WinRunner Tutorial xvii

X

- XR_GLOB_FONT_LIB 343
- XRunner, tests recorded in Analog mode 103

Z

- zipping WinRunner tests 124