WinRunner®

Testing Terminal Emulator Applications

Version 4.0

Testing Terminal Emulator Applications

© Copyright 1994 - 1997 by Mercury Interactive Corporation

All rights reserved. All text and figures included in this publication are the exclusive property of Mercury Interactive Corporation, and may not be copied, reproduced, or used in any way without the express permission in writing of Mercury Interactive. Information in this document is subject to change without notice and does not represent a commitment on the part of Mercury Interactive.

Patents pending

XRunner, WinRunner, and LoadRunner are registered trademarks of Mercury Interactive Corporation. TestDirector, TestSuite, Visual Testing, SMARTest, RapidTest, TSL and Context Sensitive are trademarks of Mercury Interactive Corporation.

This document also contains Registered Trademarks, Trademarks and Service Marks that are owned by their respective companies or organizations. Mercury Interactive Corporation disclaims any responsibility for specifying which marks are owned by which companies or organizations.

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

Mercury Interactive Corporation
470 Potrero Avenue
Sunnyvale, CA 94086
Tel. (408) 523-9900
Fax. (408) 523-9911

WRTEUG4.04/01

Table of Contents

Chapter 1: Introduction 1
About Testing Terminal Emulator Applications 1
Recording Test Scripts 2
Synchronizing Test Execution 3
Checking Your Application 3
Context Sensitive Testing with WinRunner\TE 4
Learning the Application with BMS Files 4
Analyzing Results 4
Using Default Command Softkeys 5
Typographical Conventions 6

Chapter 2: Synchronizing Test Execution 9
About Synchronizing Tests 9
Waiting for a Response from the Host 10
Waiting for a Specific String 10
Waiting for a Specific Field 11
Setting the Synchronization Time 11

Chapter 3: Checking Text 15
About Checking Text 15
Setting Text Checkpoints While Recording 16
Setting Checkpoints Automatically 17
Using Filters When Checking Text 19
Reading Text from the Screen 23
Searching for Text 23

Chapter 4: Checking GUI Objects 25
About GUI Checkpoints 25
Checking Selected Objects in an Application Screen 26
Checking All Fields in a Screen with Default Checks 27
Screen Checks dialog box 28
Field Checks dialog box 28

Chapter 5: Context Sensitive Testing with WinRunner/TE 31
About Context Sensitive Testing..31
Physical Descriptions 33
Logical Names 34
Object Classes for Terminal Emulators 34
Attributes 35
Changing the Way User Operations Are Recorded............................36

Chapter 6: Learning the Application with BMS Files..........................39
About Learning the Application with BMS Files................................39
Learning the Application the First Time...40
Relearning the Application 40

Chapter 7: Analyzing Results of Text Checkpoints.............................45
About Viewing Test Results..45
Viewing Results of a Text Checkpoint 45
Viewing Differences 47
Filtering Text Comparison Results 47

Index...49

1

Introduction

Welcome to WinRunner/TE, Mercury Interactive's automated software testing tool for terminal emulators. You can use WinRunner/TE to test mainframe, AS/400, and VAX/HP/UNIX applications running on 3270, 5250, and VT100 protocol terminal emulators, respectively.

This chapter describes:

- Recording test scripts
- Synchronizing test execution
- Checking your application
- Context Sensitive Testing with WinRunner\TE
- Learning the Application with BMS Files
- Analyzing Results of Text Checkpoints
- Using default command softkeys

About Testing Terminal Emulator Applications

When you use WinRunner/TE, you work in WinRunner's Context Sensitive recording mode. In this mode, WinRunner/TE records the operations you perform in the context of the screens, fields, and PF keys of your mainframe, AS/400, or VT100 application.

As you work with your application, WinRunner inserts TSL statements representing your actions into a test script. Among these statements are the checkpoints that define the success criteria for your test.

WinRunner/TE distinguishes between the window of the terminal emulator and screens in the host application. For the purposes of testing, the terminal emulator window, usually referred to here as “window,” consists of the frame and menus of the terminal emulator itself. This window remains constant throughout each terminal emulator session.

The screen refers to the area of the window in which the application appears. Each time the host responds to user input to the application, the screen changes.

This guide explains how to use WinRunner/TE to test mainframe, AS/400, and VT100 applications running on terminal emulators. We recommend you read the *WinRunner User’s Guide* before you read this guide. This will give you an overview of how to use WinRunner to test your application, and an explanation of WinRunner terms.

Recording Test Scripts

A test script consists of statements coded in Mercury’s Test Script Language (TSL). These statements are generated automatically in Record mode, in response to AUT input. You can also program them manually. You can mix recorded and programmed statements in the same test script.

The following is a sample of a recorded WinRunner/TE test script. The user presses the Enter key in the first screen of an application. WinRunner waits for the screen to change, and the user types the name “Minnie” in the appropriate field. The recorded statements show how WinRunner ensures that input is directed to the correct window. The comment (#) lines describe the statements.

```
# Activate the Terminal Emulator window
win_activate ("RUMBA - DEMO");

# Press the Enter key
TE_send_key (TE_ENTER);

# Wait for the next screen to refresh
TE_wait_sync();

# Direct input to the Logon screen
set_window("LOGON");
```

```
# Type in the user id ("Minnie")
TE_edit_field("USERID","Minnie");
```

Additional information on TSL is contained in the *TSL Online Reference*. You access the *TSL Online Reference* from the Help menu. You can also display information on any TSL function, in WinRunner, by placing the cursor on the function and pressing the Shift and F1 keys. The information is displayed in a Help window.

Synchronizing Test Execution

The tests that you create with WinRunner/TE run reliably every time. During a test run, execution is delayed until the application is ready to receive new input from the host. Synchronization points are recorded automatically. You can also add synchronization points through recording or programming. For more information, refer to Chapter 2, "Synchronizing Test Execution."

Checking Your Application

WinRunner verifies the behavior of your application by comparing the expected results, captured when you created your test, to the actual results when you ran the test in Verify mode.

You can use two different kinds of checkpoints to verify your application:

Text Checkpoints

You use Text checkpoints in order to compare on-screen text *according to its physical location on the screen*. WinRunner/TE can capture the entire screen of the active terminal emulator window, or only the portion of the screen that you specify. For more information, refer to Chapter 3, "Checking Text."

GUI Checkpoints

GUI checkpoints let you compare information about the screens and fields in your application interface *in terms of the objects rather than their on-screen location* during recording. You can check a screen's label, the number and

type of fields it contains, and attributes such as its color. For more information, refer to Chapter 4, “Checking GUI Objects.”

Context Sensitive Testing with WinRunner\TE

WinRunner/TE records your operations in terms of the objects on which you operate (such as screens and fields), and the type of operation you perform (such as pressing PF keys or typing in fields). Each object has a defined set of properties that determine its behavior and appearance. WinRunner/TE learns these properties and uses them to identify an locate GUI objects during a test run. For more information, refer to Chapter 5, “Context Sensitive Testing with WinRunner/TE.”

Learning the Application with BMS Files

Before you can begin Context Sensitive testing, WinRunner/TE must learn the properties of each object in your application. If you are testing a 3270 mainframe application, you can learn your application directly from a BMS file containing descriptions of the screens and fields in your application. For more information, refer to Chapter 6, “Learning the Application with BMS Files.”

Analyzing Results

After you execute a test, you can view a report of all the major events that occurred during the test run in order to determine its success or failure. You can view the expected and actual results through the WinRunner Report window.

If a mismatch is detected during a verification run, you can also view a file showing the differences between the expected and actual results. For more information, Chapter 7, “Analyzing Results of Text Checkpoints.”

Using Default Command Softkeys

Several WinRunner/TE commands can be activated using softkeys. WinRunner/TE reads input from softkeys even when the WinRunner/TE window is not the active window on your screen, or when it is minimized.

The default softkey configurations for WinRunner/TE are described in the tables below.

WinRunner Terminal Emulator Softkeys

The following table shows the softkeys that are unique to this version of WinRunner. These softkeys are for operations relating to mainframe application windows only.

Command	Softkey for 3270	Softkey for 5250	Softkey for VT100
CHECK PARTIAL TEXT	PgDown	Left Ctrl+F3	Left Ctrl+F3
CHECK TEXT	PgUp	Left Ctrl+F1	Left Ctrl+F1
GET TEXT	Left Ctrl+End	Left Ctrl+F5	Left Ctrl+F8
EXCLUDE FILTER	Left Alt+PgDown	Left Ctrl+F6	Left Ctrl+F6
INCLUDE FILTER	Right Alt+PgDown	Left Ctrl+F7	Left Ctrl+F7
WAIT STRING	End	Left Ctrl+F4	Left Ctrl+F4
WAIT SYNC	Left Ctrl+PgDown	Left Ctrl+F2	Left Ctrl+F2

Standard WinRunner Softkeys

The following table shows the default softkeys for standard WinRunner functions. Note that the default configurations for these softkeys are unique to WinRunner/TE.

Command	Softkey for 3270	Softkey for 5250	Softkey for VT100
RUN FROM ARROW	Left Ctrl+ 7	Left Ctrl+ 7	Left Ctrl+ 7
CHECK GUI	Right Ctrl+2	Right Ctrl+2	Right Ctrl+2
CHECK BITMAP AREA	Left Ctrl+2	Left Ctrl+2	Left Ctrl+2
CHECK BITMAP	Left Ctrl+PgUp	Right Ctrl+0	Right Ctrl+0

Command	Softkey for 3270	Softkey for 5250	Softkey for VT100
INSERT FUNCTION FROM LIST	Left Alt+7	Left Alt+7	Left Alt+7
MARK LOCATOR	Right Ctrl+6	Right Ctrl+6	Right Ctrl+6
OBJECT INSERT FUNCTION	Left Alt+8	Left Alt+8	Left Alt+8
PAUSE	PAUSE	PAUSE	PAUSE
GET TEXT AREA	Left Ctrl+1	Left Ctrl+1	Left Ctrl+1
RECORD	Scroll Lock	Left Alt+2	Scroll Lock
RUN FROM TOP	Left Ctrl+5	Left Ctrl+5	Left Ctrl+5
STEP	Left Ctrl+6	Left Ctrl+6	Left Ctrl+6
STEP INTO	Left Ctrl+8	Left Ctrl+8	Left Ctrl+8
STEP TO CURSOR	Left Ctrl+F9	Left Ctrl+F9	Left Ctrl+F9
STOP	Left Ctrl+3	Left Ctrl+3	Left Ctrl+3
WAIT BITMAP AREA	Left Ctrl+4	Left Ctrl+4	Left Ctrl+4
WAIT BITMAP	Left Ctrl+0	Left Ctrl+0	Left Ctrl+0
GUI CHECK LIST	Right Ctrl+F12	Right Ctrl+F12	Right Ctrl+F12
GET TEXT OBJECT	Left Ctrl+9	Left Ctrl+9	Left Ctrl+9

Softkey assignments are configurable. If the application you are testing uses one of the default softkeys preconfigured for WinRunner/TE, you can redefine the softkey by modifying the *wrun.ini* configuration file. For details, refer to the *WinRunner User's Guide.*

Typographical Conventions

This book uses the following typographical conventions:

Bold — **Bold** text indicates function names and the elements of the functions that are to be typed in literally.

Italics — *Italic* text indicates variable names.

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

2

Synchronizing Test Execution

WinRunner/TE provides complete synchronization between the host and the application under test (AUT) during test execution. Synchronization ensures that test execution is delayed until the application is ready to receive new input. This prevents incidental differences in host response time from affecting successive test runs.

This chapter describes:

- ➤ Waiting for a response from the host
- ➤ Waiting for a specific string
- ➤ Waiting for a specific field
- ➤ Setting the synchronization time

About Synchronizing Tests

When using a terminal emulator, many factors can affect the speed of operation and therefore interfere with test execution. Host response time varies with load on the system. The screen refresh rate of your terminal can also vary. WinRunner/TE provides different types of synchronization points to pace test execution with the system. These points are inserted into the test script automatically, or by either programming or recording.

Waiting for a Response from the Host

In recording, WinRunner/TE automatically generates the following statement each time the terminal emulator waits for a response from the host:

TE_wait_sync ();

During a test run, this statement ensures that test execution is delayed until the host responds and the new screen is completely redrawn.

To generate the **TE_wait_sync** statement in your test script during recording, press the WAIT SYNC softkey.

Waiting for a Specific String

Using the **TE_wait_string** function, you can instruct WinRunner to wait for a specific string to appear on the screen before continuing test execution. You can specify an area of the screen, or WinRunner can search the entire screen for the string.

To record a TE_wait_string statement in your test script:

1. During recording, press the WAIT STRING softkey. WinRunner is minimized to an icon and a dialog box displays instructions for capturing the string.
2. Enclose the text you want WinRunner to look for during test execution in a rectangle: press and hold down the left mouse button and drag the mouse until the rectangle encloses the desired area.
3. To capture the string, click the right mouse button. WinRunner is restored and a **TE_wait_string** statement with the following syntax is inserted into your test script:

TE_wait_string (*string*, **[** *start_column*, *start_row*, *end_column*, *end_row* **],**
[*timeout* **]);**

The *string* parameter is the text enclosed in the rectangle. If the text you captured exceeds one line, *string* includes the first line only. The *start_column* and *start_row* parameters indicate the column/row at which the captured text starts. The *end_column* and *end_row* parameters represent the column

and row, respectively, at which the text ends. The *timeout* parameter is the number of seconds that WinRunner waits for the specified string to appear before continuing test execution.

The example that follows shows the statement recorded when the text of a menu option is captured using the WAIT STRING softkey:

```
TE_wait_string ("Open the mail", 8, 4, 20, 4, 60);
```

The first parameter, “Open the mail”, is the string that WinRunner searches the screen for; WinRunner will look for this string in row 4, columns 8 through 20. The default timeout is 60 seconds.

When you program this statement, you can eliminate the coordinates. In this case, WinRunner searches the entire screen for the specified string. You can also change or eliminate the *timeout* parameter. If there is no *timeout* parameter, then the system timeout is used.

Waiting for a Specific Field

Using the **TE_wait_field** function, you can instruct WinRunner to wait for a specific field to appear on the screen before continuing test execution. When the field appears, WinRunner resumes test execution. The syntax for this function is:

TE_wait_field (*field_logical_name***,** *content***,** *timeout* **);**

The *field_logical_name* parameter is the name of the field that WinRunner will wait for. The *content* parameter is the string contained in the field. The timeout is the number of seconds that WinRunner waits for the specified field to appear before continuing test execution.

Setting the Synchronization Time

Two factors that can affect proper test execution are the response time of the host and the screen refresh rate of your terminal. The following functions allow you to configure WinRunner to handle these variations.

Changing the Screen Refresh Time

The **TE_set_refresh_time** function determines how long WinRunner waits for the screen to refresh after the host has responded.

The syntax for this function is:

TE_set_refresh_time (*time* **);**

The default *time* is 1 second. You can increase this if needed to ensure that WinRunner waits until the screen is completely redrawn before continuing test execution.

Changing the Timeout

The **TE_set_timeout** function determines the maximum amount of time that WinRunner waits for a response from the host before continuing test execution.

This statement has the following syntax:

TE_set_timeout (*timeout* **);**

The default *timeout* is 60 seconds. You can modify this if needed.

Setting the System Synchronization Time

The **TE_set_sync_time** function determines the minimum number of seconds that WinRunner waits for the host to respond. WinRunner uses this information to determine that synchronization has been achieved before continuing test execution.

This statement has the following syntax:

TE_set_sync_time (*time* **);**

Getting the System Synchronization Time

The **TE_get_sync_time** function returns the minimum number of seconds that WinRunner will wait for the host to respond. WinRunner uses this information in order to determine that synchronization has been achieved before continuing test execution.

This statement has the following syntax:

TE_get_sync_time (*time***);**

3

Checking Text

You can use WinRunner/TE to check the text in the screen of your mainframe, AS/400, or VAX/HP/UNIX application.

This chapter describes:

- Setting text checkpoints while recording
- Setting checkpoints automatically
- Using filters when checking text
- Reading text from the screen
- Searching for text

About Checking Text

WinRunner/TE provides different methods of checking the text in your host application screen. You can:

- capture all or part of the screen contents while recording a test
- instruct WinRunner/TE to automatically capture all or part of the screen contents of the active terminal emulator window

While creating a test, you indicate the text that you want to check. WinRunner inserts a checkpoint in the script, captures the specified text, and stores it in the expected results directory (*exp*) of the test. When you run the test, WinRunner recaptures the text and compares it to the expected text captured earlier. You can view both the expected and the actual test results. In the case of a mismatch, you can also view any differences between them.

You can also use WinRunner to read text from a selected portion of the screen and store it in a variable. The screen coordinates of the text you indicated are inserted into the test script. You could use this feature, for example, to change the logical flow of a test run during test execution according to the text found in the indicated area.

Setting Text Checkpoints While Recording

You can capture the entire contents of the terminal emulator window for comparison. Alternatively, you can select a specific portion of the screen for text capture. All captured text is stored as ASCII text. You can view these files through the WinRunner Report window.

Checking a Full Screen

Use a full-screen text checkpoint to capture the entire contents of the active terminal emulator window.

To capture the contents of the screen:

1 During recording, make sure that the terminal emulator window you want to check is active.

2 Press the CHECK TEXT softkey. A **TE_check_text** statement is recorded in your test script.

The entire contents of the active terminal emulator window are captured (even if not all of the text is visible in the window). A **TE_check_text** statement such as the following is inserted into the test script:

```
TE_check_text ("Trm1");
```

The default name that WinRunner assigns to the first incidence of a full-screen text checkpoint in a test script is called Trm1. The text is stored as an ASCII file in the expected results directory of the test.

When you run the test, WinRunner compares the text currently displayed on the screen with the expected text captured earlier (the contents of the file Trm1, stored in the expected results directory). In the event of a mismatch, WinRunner captures the actual text and generates a difference file that

shows the discrepancy between the expected and the actual results. Both files are stored in the current verification results directory.

Checking a Partial Screen

Use partial text checkpoint when you want to capture only part of the text on the screen.

To capture text in an area of the screen:

1. Press the CHECK PARTIAL TEXT softkey. WinRunner is minimized to an icon and a dialog box displays instructions for capturing the text.
2. Enclose the text to be captured within a rectangle. Press and hold down the left mouse button and drag the mouse until the rectangle encloses the desired area.
3. Click the right mouse button: WinRunner is restored and a **TE_check_text** statement such as the following is inserted in the test script:

```
TE_check_text ("Prt1", 51, 13, 60, 13);
```

The example shows the statement recorded when the text in line 13, columns 51 through 60 is captured. The default file name "Prt1" indicates the first incidence of captured partial text in any test script.

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

Setting Checkpoints Automatically

You can instruct WinRunner/TE to capture the contents of the active terminal emulator window each time a new screen appears. The three main options for automatic text checkpoints are:

- Check full screen
- Check partial screen
- Check partial screen using the previous “check partial screen” coordinates

Checking Full Screens

When full screen automatic text check is active, all of the text in the active window is captured each time a new screen is displayed.

To activate full screen automatic text check, execute the following statement in your test script:

TE_set_auto_verify (ON **);**

Each time a new text screen is displayed in the window, a **TE_check_text** statement like the following is automatically inserted into the test script.

TE_check_text ("Trm1" **);**

To deactivate automatic text check, execute the following statement:

TE_set_auto_verify (OFF **);**

Checking Partial Screens

When partial screen automatic text check is active, the text in the specified area of the active window is captured each time a new screen appears.

To activate partial screen automatic text check, program and execute a statement with the following syntax in your test script:

TE_set_auto_verify (*ON, start_column, start_row, end_column, end_row***);**

ON activates automatic check; ***start_column*** indicates the column at which the captured text starts; ***start_row*** indicates the row at which the captured text starts; and ***end_column*** and ***end_row*** represent the column and row, respectively, at which the text ends.

The example below shows the statement you would execute to automatically check the text in columns 22 through 31, rows 10 through 14.

```
TE_set_auto_verify (ON, 22, 10, 31, 14);
```

Each time a new screen appears in the window, a **TE_check_text** statement similar to the following is automatically inserted into the test script.

```
TE_check_text ("Prt1", 22, 10, 31, 14);
```

To deactivate automatic partial text check, execute the following statement:

```
TE_set_auto_verify (OFF);
```

Checking Partial Screens Using Previous Coordinates

When you choose the first/last partial text option, the coordinates for the partial screen automatic check are taken from a previous **TE_check_text** statement in the test run.

To activate first/last partial screen automatic text check, execute a statement with the following syntax in your test script:

TE_set_auto_verify (ON**,** FIRST|LAST**);**

If you use the FIRST parameter, the coordinates for the automatic partial screen check will be taken from the first **TE_check_text** statement in the test run. If you use the LAST parameter, the coordinates will be taken from the last **TE_check_text** statement in the test run. The coordinates are updated during the test run with each **TE_check_text** statement.

Note that if there is no **TE_check_text** statement in the test script, then the entire screen is captured.

To deactivate first/last partial screen automatic text check, execute the following statement in your test script:

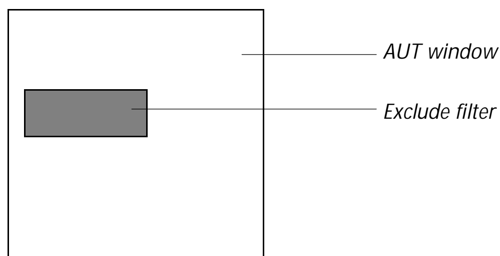
```
TE_set_auto_verify (OFF);
```

Using Filters When Checking Text

WinRunner lets you use filters to include or exclude regions of a terminal emulator window when checking text. In cases where you do not want to check an entire window, you can define parts of the window that will be filtered during the comparison. You can use two types of filters: ***exclude*** and ***include***.

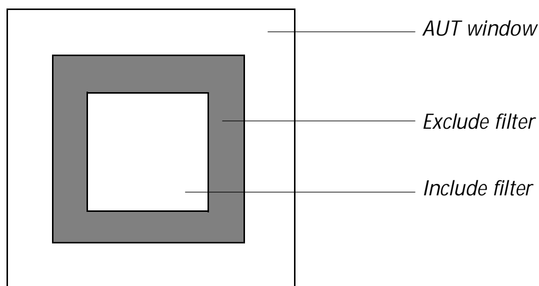
Exclude and Include Filters

An exclude filter defines the area to be ignored during the comparison. For example, you can create an exclude filter on a region of a window containing the current date and time.



AUT window with exclude filter

An include filter is used in combination with an exclude filter. In the diagram below, the white areas are included in the comparison and the shaded area is excluded. This is achieved by defining an exclude filter and then defining a smaller include filter on top of it. The result is a "ring" that is excluded from comparison.



AUT window with exclude filter and include filter

Note that when you combine exclude and include filters, the order in which the filters are activated in the test script determines the actual area of interest. For example, if an exclude filter that fully or partially overlaps an include filter is activated after the include filter, the overlapped region is excluded from the area of interest.

Note: You can set up to 256 filters using **TE_set_filter**.

Creating Filters

You use the EXCLUDE FILTER and INCLUDE FILTER softkeys to create a filter during recording.

To create a filter by recording:

1. During recording, press the appropriate softkey (FILTER EXCLUDE or FILTER INCLUDE). WinRunner is minimized to an icon and a dialog box displays instructions for defining the filter area.
2. Enclose the area to be filtered inside a rectangle. Press and hold down the left mouse button and drag the mouse until the rectangle encloses the desired area.
3. To record the filter, click the right button.

   WinRunner is restored. The filter is added to the test's db directory and a **TE_set_filter** statement is inserted into your test script.

   The following example shows what WinRunner records when an exclude filter is defined on row 23, columns 1 through 30 of the active terminal emulator window.

   ```
   TE_set_filter ("Filter0",1, 23, 30, 23, EXCLUDE);
   ```

   When a **TE_set_filter** statement is executed during a test run, the filter is activated. For more information on **TE_set_filter**, refer to *TSL Online Reference*.

Deactivating and Deleting Filters

When you deactivate an existing filter, it remains in the test's db directory but is inactive for the test. To deactivate a filter, execute a statement with the following syntax in your test script:

TE_reset_filter (*filter_name***);**

You can also define the filter to be deactivated using its coordinates and type, instead of its name. Execute a statement with the following syntax:

TE_reset_filter (*start_column, start_row, end_column, end_row,*
EXCLUDE || INCLUDE **);**

To deactivate all active filters, execute the following statement:

TE_reset_all_filters();

To delete a filter from the test database, execute a statement with the following syntax in your test script:

TE_delete_filter (*filter_name***);**

Creating and Activating Filters Separately

In some cases you may wish to create a filter and store it in the test's db directory for later use. Use the **create_filter** function to create a filter; activate it by executing a **TE_set_filter** statement containing only the name of the filter.

To create a filter, execute a statement with the following syntax in your test script:

TE_create_filter (*filter_name, start_column, start_row, end_column, end_row*,
EXCLUDE | INCLUDE **);**

The *filter_name* can be up to 16 characters long.

To activate a filter, execute the following statement in the script:

TE_set_filter (*filter_name***);**

The *filter_name* must be the name of an existing filter for the current test.

Reading Text from the Screen

Using the **TE_get_text** function, you can instruct WinRunner/TE to read the text in a specified area of the screen and store it in a variable. During recording, you use the mouse to define the area of the screen to be read. You can also program the **TE_get_text** function.

To read text from the screen:

1. Make sure that you are in recording mode and that the terminal emulator window you want to read from is in focus.
2. Press the GET TEXT softkey. WinRunner is minimized to an icon and a dialog box displays instructions for capturing the string.
3. Enclose the text to be read within a rectangle. Press and hold down the left mouse button and drag the mouse until the rectangle encloses the desired area.
4. To capture the string, click the right mouse button. WinRunner is restored. A **TE_get_text** statement is inserted in the test script. This statement has the following syntax:

   t = TE_get_text (*x1, y1, x2, y2* **);**

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

Each new line of the on-screen text that is captured is preceded by the characters "\n" in the variable. The following example shows how two lines of on-screen text appear in the variable *t*:

```
t = "Fill in your User ID and press Enter \n(Your password will not appear
    when you type it)"
```

Searching for Text

You can search for text in a terminal emulator screen using the **TE_find_text** function. This function looks for a specified text string and returns its location on the screen as an *x* coordinate and a *y* coordinate. Using an optional parameter, you can restrict the search to a rectangular area of the screen that you define using pairs of *x*, *y* coordinates.

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

TE_find_text (*string*, *out_x_location*, *out_y_location* [x_1, y_1, x_2, y_2]);

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

4

Checking GUI Objects

WinRunner/TE sees the terminal emulator application window as a screen containing fields. You can capture information about each screen and its contents and store the information as a basis for comparison.

This chapter describes:

- Checking selected objects in an application screen
- Checking all fields in an application screen
- The Screen Checks dialog box
- The Field Checks dialog box

About GUI Checkpoints

WinRunner/TE enables you to compare information about the user interface of your mainframe, AS/400, or VAX/HP/UNIX application between versions. You can use GUI checkpoints to check the label of a screen, as well as the number and type of fields within the screen. For example, you can check the content, type, and location, for any field.

To create a GUI checkpoint, you select a screen or field and define the checks you want to perform. You can use the default checks or define custom checks. Information about the screens and fields as well as the checks is saved in a checklist. WinRunner captures the current state of these objects and saves this information as expected results. A GUI checkpoint is automatically inserted into the test script. This checkpoint appears as an **obj_check_gui** or **win_check_gui** statement.

When you run the test, WinRunner compares the current state of the application to the expected results, the information is captured and

compared to the expected results. If the expected results and the current results do not match, the GUI checkpoint fails. The results of the checkpoint can be viewed in the WinRunner Report dialog box.

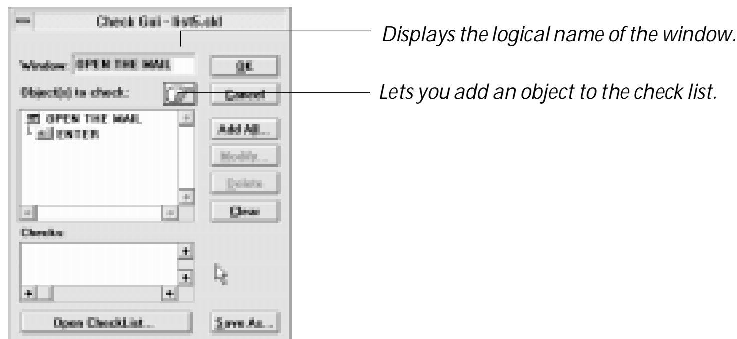
Note: The procedure for creating a checklist for fields and screen differs slightly from those for other GUI objects. For information on checking standard Windows objects, see the *WinRunner User's Guide*.

Checking Selected Objects in an Application Screen

You can capture information about one or more fields in a screen.

To capture GUI data of selected objects for comparison:

1. Select Check GUI > Object/Window from the Create menu. The WinRunner window is minimized to an icon, the mouse pointer turns into a pointing hand, and a popup window appears on the screen.
2. Activate the screen you want to check. The Check GUI dialog box opens.



Click on the screen or field you want to check. The selected object flashes and is inserted into a checklist according to the default check(s) for the object class. Each object you click on is added to the checklist.

3. Press the right mouse button to stop the add operation and restore the mouse pointer to its original shape. The Check GUI dialog box appears on

the screen with the default checks. To choose different checks, select the object in the list and click the Modify button. The appropriate check dialog box appears. For more information, see the sections “Screen Checks dialog box” and “Field Checks dialog box.”

4 Mark the check(s) to perform and select OK to close the checks dialog box and return to the Check GUI dialog box.

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

WinRunner captures all the objects in the GUI checklist, inserts a **check_gui** statement in your test script, and resumes recording.

Checking All Fields in a Screen with Default Checks

You can create a checklist containing all the fields in the selected screen. During test execution, WinRunner compares the expected and actual results for all the fields in the current screen as well as the default check for the field class (field_content).

To capture all the fields in a screen:

1. Select Check GUI > Object/Window from the Create menu. The WinRunner window is minimized to an icon, the mouse pointer turns into a pointing hand, and a popup window appears on the screen.
2. Activate the screen you want to check. A popup window asks whether you want to check all the objects in the window. Select Yes.
3. Click on the pointing hand button. The mouse pointer turns into a pointing hand.
4. The Add All dialog box opens.
5. Click OK button to add all the fields in the screen to the checklist. WinRunner generates a new checklist containing the objects specified. This may take several seconds.
6. Click OK to save the checklist and close the dialog box.
7. WinRunner captures the GUI information and stores it in the test’s expected results directory. The WinRunner window is restored and a **check_GUI** statement is inserted into the test script.

Screen Checks dialog box

The Screen Checks dialog box lets you modify the GUI checklist for a screen.

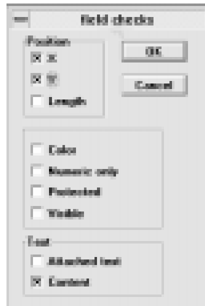
number of protected fields: checks the number of protected fields in the screen (default check).

number of input fields: checks the number of unprotected fields in the screen (default check).

label: checks the label (title) of the screen.

Field Checks dialog box

The Field Checks dialog box lets you modify the GUI checklist for a field.



x and **y**: checks the *x* and *y* coordinates of the top left corner of the field, relative to the screen origin (default checks).

length: checks the length of the field, in characters.

color: checks the color of the field.

numeric only: checks whether the field is numeric only.

protected: checks whether the field is protected.

visible: checks whether the field is visible.

attached text: checks the attached text of the field.

content: checks the content of the field (default check).

5

Context Sensitive Testing with WinRunner/TE

You can use WinRunner's Context Sensitive features to test mainframe, AS/400, and VAX applications running on terminal emulators for 3270, 5250, and VT100. For general information on Context Sensitive testing with WinRunner, see the Understanding the GUI Map section of the ***WinRunner User's Guide***.

This chapter describes:

- Physical descriptions
- Logical names
- Object classes
- Attributes
- Changing the record methods

About Context Sensitive Testing

Context Sensitive testing ensures that non-essential changes in your applications do not affect test execution. WinRunner/TE can handle changes in window size between testing sessions, or modifications in the positioning of fields in an application screen. WinRunner/TE records your operations in terms of the objects on which you operate (such as screens and fields), and the type of operation you perform (such as pressing PF keys or typing in fields). It ignores the physical location of objects on the screen.

To perform Context Sensitive testing, WinRunner must uniquely identify each object and be able to locate it in the application under test (AUT).

During Context Sensitive testing, WinRunner learns an accurate description of each object as it is identified by the AUT. If you have access to the BMS files of your application, WinRunner/TE can learn your application by reading these files directly. Refer to Chapter 6, “Learning the Application with BMS Files” for more information. Otherwise, WinRunner learns a description of each object using RapidTest Script Wizard, recording, or the GUI Map Editor. For more information on these methods, refer to the *WinRunner User’s Guide*.

The description of each GUI object (called the *physical description)* contains a detailed list of attributes. WinRunner places this list in a GUI file. In the test script, WinRunner uses an intuitive *logical* name for each object (as it appears in the application).

The following example illustrates the connection between the logical name and the physical description. Assume that you record a test in which you type your user ID in the Login screen of your application.

WinRunner/TE learns the actual description, or list of attributes, of both of the objects you operated on:

Screen	*{class:mic_if_win, label:VIRTUAL MACHINE/SYSTEM PRODUCT, mic_if_handles_windows:1}*
Field	*{class:field, attached_text:"USERID"}*

WinRunner identifies the screen as the class *mic_if_win* (a host application window), and its label as VIRTUAL MACHINE/SYSTEM PRODUCT;

mic_if_handles_windows is an internal attribute used by WinRunner. The USERID field is recognized as the class *field* with the attached text "USERID". In the test script, WinRunner inserts intuitive logical names for the objects. If you start recording and type the user name "Carmen", the script segment might look like this:

```
set_window ("VIRTUAL MACHINE/SYSTEM PRODUCT");
TE_edit_field("USERID","Carmen");
```

When the test is run, WinRunner reads the logical name of each object from the script and refers to its physical description in the GUI map file. It uses this description to find the object in the AUT.

Physical Descriptions

The physical description of an object contains a list of attribute–value pairs, as follows:

{*attribute1:value1, attribute2:value2, attribute3:value3, ...*}

For example, the description of the "Login" screen presented above contains three attributes, listed below together with their values:

class: mic_if_win
label: VIRTUAL MACHINE/SYSTEM PRODUCT
mic_if_handles_windows: 1

WinRunner always learns the *class* attribute. This indicates the type of the GUI object, such as the terminal emulator window, host application screen, or field. For each class, WinRunner learns a set of default attributes. For more information on attributes that are unique to WinRunner for terminal emulators, see "Attributes" in this chapter. For information on other attributes used by WinRunner, see the *WinRunner User's Guide.*

Note that WinRunner learns the physical description of an object in the context of the window in which it appears. This creates a unique physical description for each object.

Logical Names

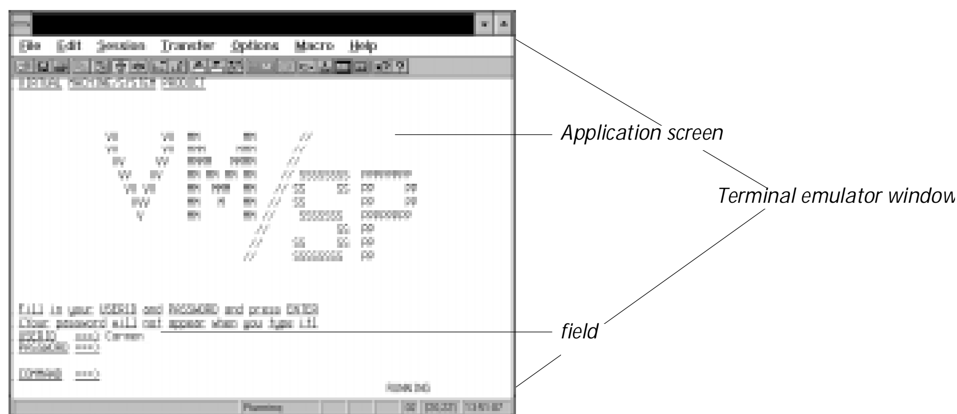
The logical name is the name WinRunner uses for objects in the test script. Once the name is assigned, you can modify it in the GUI map file.

The logical name assigned to an object depends on the *class* of the object. For example, the logical name of a window is the value of its *label* attribute. The logical name of a field is the value of its *attached_text* attribute.

Object Classes for Terminal Emulators

WinRunner/TE identifies two types of objects for terminal emulators: *screens* and *fields*. The screen is the application area. It changes each time input is received from the host. Fields include unprotected fields, which can receive input, and protected fields which contain fixed text.

WinRunner/TE also identifies the window of the terminal emulator, the outer frame of the terminal emulator including its menus, scrollbar, and buttons. The class attribute of this window is always mic_if_window. For more information on this class, see the *WinRunner User's Guide.*



Attributes

The following table shows the attributes for application screens and fields. For a full list of attributes for all standard Windows objects, see the *WinRunner User's Guide*.

Screens

A screen can have the following attributes:

Attribute	Description
class	The prime attribute that WinRunner uses to identify the type of GUI object. All screens belong to the class “mic_if_win”.
label	The title of the screen. If there is no title, WinRunner assigns a unique number.
protected_fields_number	The number of protected fields in this screen.
input_fields_number	The number of unprotected fields in this screen.
id	A number that WinRunner uses to identify the screen.
mic_if_handles_windows	An internal attribute that WinRunner uses. The value of this attribute is always 1.

Fields

A field can have the following attributes:

Attribute	Description
class	The prime attribute that WinRunner uses to identify the type of GUI object. All fields belong to the class “field”.
attached_text	The text that is closest to the field.
protected	A value that indicates whether the field is protected. This value is “yes” if the field is protected; otherwise it is “no”.
visible	A value that indicates whether the contents of the field can be seen: 1 if they are visible, 0 if not.

Attribute	Description
numeric_only	A value that indicates whether the field is numeric. This value is “yes” if the field is numeric; otherwise the value is “no”.
id	A number that WinRunner uses to identify the field.
x	The x coordinate of the top left corner of a field, relative to the window origin.
y	The y coordinate of the top left corner of a field, relative to the window origin.
length	The length of the field, in characters.
color	A value indicating the color of the field. This can be 0, 1, 2, or 3, depending on the terminal emulator’s color definitions.

Changing the Way User Operations Are Recorded

By default, WinRunner/TE records operations on screens, fields, and PF keys using functions such as **TE_edit_field** and **TE_send_key**. This record method also enables the use of GUI checkpoints, the GUI Map Editor, and other WinRunner Context Sensitive features. The “field” method is available for 3270 and 5250 protocol terminal emulators only.

A second record method is also available. (For VT100 terminal emulators this is the sole record method available.) When the “position” method is used, WinRunner/TE records keyboard and mouse input only; operations on objects in your application are recorded as **type**, **win_mouse_click**, and **win_mouse_drag** statements. Context Sensitive features are not available.

Note: The record method (field or position) is not the same as the WinRunner record mode (Context Sensitive or Analog). Note also that WinRunner/TE must always be in Context Sensitive record mode.

You use the **TE_set_record_method** function to change the record method. This function has the following syntax:

TE_set_record_method (*method* **);**

The *method* can be one of the following:

- FIELD_METHOD, or (2) (the default): enables full Context Sensitive recording.
- POSITION_METHOD, or (1): keyboard and mouse input only is recorded.

Note that the current record method remains valid until you change it, even after you exit WinRunner/TE and start it again.

6

Learning the Application with BMS Files

The Learn BMS Files feature can teach WinRunner/TE your 3270 mainframe application by inserting information about screens and fields directly into a GUI map file. This chapter describes:

- Learning the application the first time
- Relearning the application

About Learning the Application with BMS Files

If you have access to the BMS file of your 3270 mainframe application, you can use the Learn BMS Files feature. This feature enables WinRunner/TE to learn your application directly from a BMS file containing descriptions of the screens and fields in your application. When you use Learn BMS File, WinRunner learns these descriptions and inserts them into a GUI map file. You can change the names or descriptions as desired, as with any other GUI map file. You use the TSL function **TE_bms2gui** to learn the BMS file.

The RELEARN option lets you update the GUI map file you created earlier as your application changes during the development cycle. An interactive user interface guides you through the process. It helps you retain desired modifications to the descriptions in the GUI map file while changing others as needed.

It is recommended that you be familiar with the chapter “Chapter 5, “Context Sensitive Testing with WinRunner/TE”” as well as the Understanding the GUI Map section of the *WinRunner User’s Guide* before you begin Learn BMS Files.

Learning the Application the First Time

You use the **TE_bms2gui** function to learn (and to relearn) your BMS file. This function has the following syntax:

TE_bms2gui ("*bms_file_name", "gui_file_name", learn_mode* **);**

The *bms_file_name* parameter is the full path of the BMS file of your application. The *gui_file_name* parameter is the full path of the GUI map file in which WinRunner inserts the descriptions of the objects in your application. If no parameter is specified, the temporary GUI map file is used.

The *learn_mode* parameter determines how WinRunner/TE handles the BMS file. Use the LEARN option the first time that you learn a BMS file. Do not perform LEARN twice for the same GUI map file. Use RELEARN when you have made changes to your application and updated the BMS file. When RELEARN is specified, WinRunner compares the descriptions in the current BMS file with those in the specified GUI map file. It notifies you of any inconsistencies and allows you to make changes as desired.

To learn the BMS files, execute the **TE_bms2gui** function in a WinRunner script. In the following example, **TE_bms2gui** is used to teach WinRunner object descriptions from a BMS file called Mail_app.txt and place them into a GUI map file called Mail_1.gui:

```
TE_bms2gui ("Mail_app.txt", "Mail_1.gui", LEARN);
```

You can edit names or descriptions in the GUI map file created by **TE_bms2gui** and make any other desired changes, using the GUI Map Editor. See the *WinRunner User's Guide* for more information.

Relearning the Application

You use the RELEARN option each time you want to update the GUI map file to reflect changes in your application. RELEARN enables you to add new screens and fields to the GUI map file while maintaining or changing the names and descriptions that appear in the existing GUI map file, as desired.

To relearn a BMS file, you execute the **TE_bms2gui** function using RELEARN as the *learn_mode* parameter. For example, to relearn a BMS file called

Mail_app.txt into an existing GUI map file called Mail_1.gui, execute the following statement:

```
TE_bms2gui ("Mail_app.txt", "Mail_1.gui", RELEARN);
```

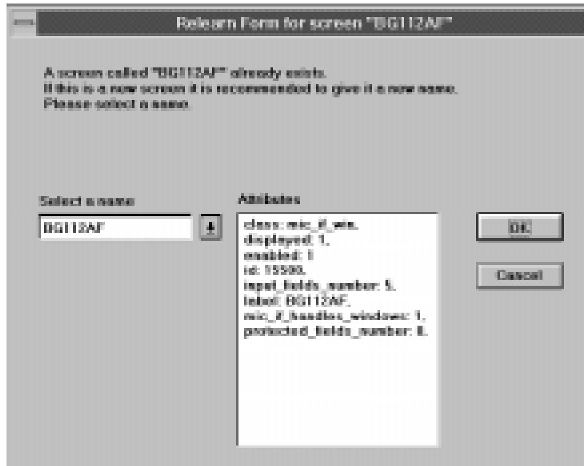
As WinRunner/TE converts the BMS file into the GUI map file, it looks for discrepancies between the BMS file learned using the LEARN option and the current file, on which RELEARN is performed. Each time it finds a mismatch, a dialog box appears on screen and asks your how to proceed.

In most cases, accepting the default option ensures that the intentional changes made to your application are reflected accurately in the GUI map file. However, WinRunner always gives you the option of changing the name of the relevant screen or field.

The following paragraphs describe the different Relearn forms that may be displayed during the RELEARN process and the options they provide.

Note: The forms are identical for fields and for screens, with the exception of the word “field” or “screen” in the relevant location.

The Object Exists in the GUI Map File with Different Attributes



WinRunner found a screen in the BMS file with the same name as a screen in the existing GUI map file, but with different attributes. The current name of the screen is displayed in the list on the left side of the Relearn dialog box. The list on the right shows all the attributes of the selected object, according to the new BMS file. By default, WinRunner updates the GUI map to include the new attributes.

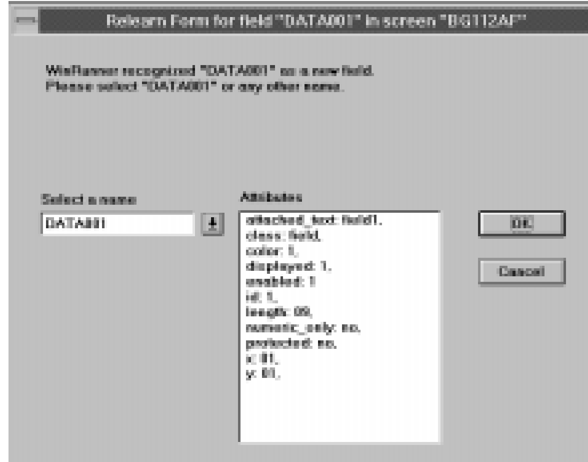
Click OK. The following message appears: “Screen BG112AF is now changed and gets new attributes”. Click OK.

To use a different name for the screen, select it from the list or type in another name.

To continue the RELEARN operation without making changes, click Cancel.

To choose a new name for the object, type it in or select the name of an existing object from the list.

This Object Is Not in the Original GUI Map File



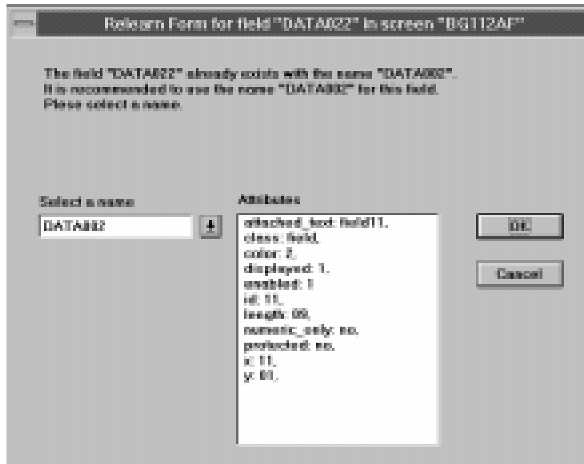
WinRunner found a field that it recognizes as a new one: no other field with the same name or attributes exists in the GUI map file. The name of the field is displayed in the list on the left side of the Relearn dialog box. The list on the right shows all the attributes of the selected field, according to the new BMS file.

By default, WinRunner adds the object to the GUI map file with the name specified. The Relearn dialog box closes and the following message appears: “WinRunner added a new field with the name “DATAOO1.”

To continue the Relearn operation without making changes, click Cancel.

To choose a new name for the object, type it in or select the name of another screen from the list.

This Object Appears in the GUI Map File with a Different Name



WinRunner found a field with the same attributes as an existing field, but with a different name. By default, WinRunner retains the original name for the field as it appears in the GUI map. This ensures that you can replay existing tests containing the original name for the field without changing them.

Click OK to retain the original name for the field. The Relearn dialog box closes and the following message appears: “WinRunner uses the existing field ‘DATA002’”.

To use the name in the new BMS file or to select a new name, select it from the list or type it in.

7

Analyzing Results of Text Checkpoints

After you execute a test, you can view a report of all the major events that occurred during the test run in order to determine its success or failure.

This chapter describes:

- Viewing results of a text checkpoint
- Viewing differences
- Filtering text comparison results

About Viewing Test Results

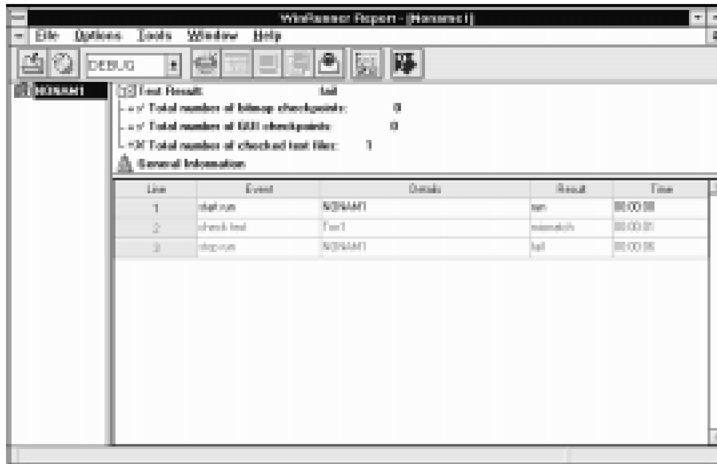
When a test run is completed, you can view detailed test results in the WinRunner Reports window. To open the window, select Reports from the Tools menu or click the Reports icon. The Report window opens and displays the results of the current test. You can view expected, debug, and verification results in the Report window. By default, the Report window displays the results of the most recently executed test run. For more information, see the *WinRunner User's Guide.*

Viewing Results of a Text Checkpoint

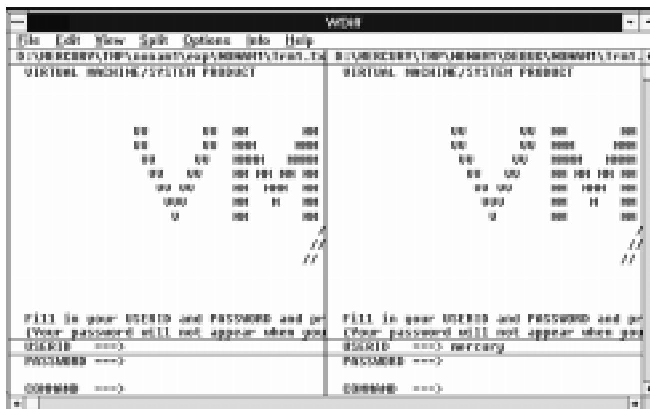
A text checkpoint compares expected and actual text in your application. You can view the expected and actual results through the Report window. If a mismatch is detected during a verification run, you can also view a file showing the differences between the expected and actual results.

To view the results of a text checkpoint:

1 In the test log, look for entries that list text comparisons in the Event column.



2 To display the results of a specific text comparison, double-click on its entry in the log or select the entry and click the Display icon. If there is no mismatch, the Expected results are displayed in a Notepad window. Following a mismatch, the expected and actual results are displayed in the WDiff utility window.



Expected results *Actual results*

Lines in the file that contain a mismatch are highlighted. The file in the first parameter of the **file_compare** function is on the left side of the window.

3 Select File > Exit to close the window.

Viewing Differences

To see the next mismatch in the captured text, select Next Diff from the View menu or press Tab. The window scrolls to the next highlighted line. To see the previous difference, select Prev Diff or press the backspace key.

Filtering Text Comparison Results

You can choose to view only the lines in the captured text that contain a mismatch. To filter text comparison results, select Hide Matching Areas from the Options. The window shows only the highlighted parts of both captured texts.

Index

A

Add All 27
attached_text attribute 35
Attributes 35

B

BMS Files 39

C

Check GUI form 26
check text softkey 16
Checking GUI objects 25–29
Checking Text 15–24
Checking text
  full screen 16
  partial screen 17
Checking text automatically 17
  full screen 18
  partial screen 18
  using previous coordinates 19
class attribute 35
color attribute 36
Context Sensitive Testing for Mainframe Applications 31–37

E

Exclude filter 20
exclude filter softkey 21

F

Field checks form 28
Filters 19–22
  exclude 20
  include 20

G

get text softkey 23
GUI Checkpoints 3

I

id attribute 35, 36
Include filter 20
include filter softkey 21
input_fields_ number attribute 35

L

label attribute 35
Learning the user interface using BMS files 39
length attribute 36
Logical names 34

M

mic_if_handles_windows attribute 35

N

numeric_only attribute 36

O

Object classes 34

P

Physical description 33
protected attribute 35

protected_fields_number attribute 35

R

Reading Text 23
Record method 36

S

Screen checks form 28
Scripts, test 2
Softkeys 5
Synchronizing Tests 9–13

T

TE_check_text function 16
TE_create_filter function 22
TE_find_text function 23
TE_get_text function 23
TE_reset_all_filters function 21
TE_reset_filter function 21
TE_set_auto_verify function 18
TE_set_filter function 21, 22
TE_set_record_method function 36
TE_wait_string function 10
TE_wait_sync function 10
Text Checkpoints 3
Text comparison
    filtering results 47
    viewing differences 47
Text, finding on screen 23
TSL 2
TSL Online reference 3

V

visible attribute 35

W

wait string softkey 10
wait sync softkey 10

X

x attribute 36

Y

y attribute 36