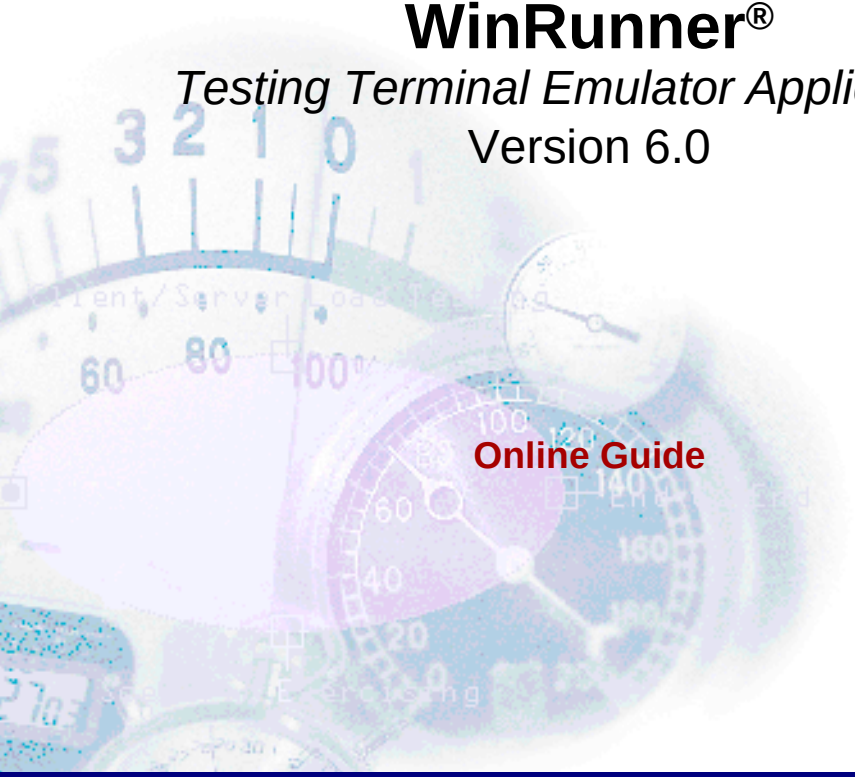
WinRunner®

Testing Terminal Emulator Applications

Version 6.0

Online Guide



Books Online

Find

Find Again

Help

Table of Contents

Introduction

Welcome to WinRunner with add-in support for terminal emulator applications. You can use WinRunner to test mainframe, AS/400, and VAX/HP/UNIX applications running on 3270, 5250, and VT100 protocol terminal emulators, respectively.

When using WinRunner to test terminal emulator applications, you work in WinRunner's Context Sensitive recording mode. In Context Sensitive mode, WinRunner records the operations you perform in the context of the screens, fields, and PF keys of your mainframe, or AS/400 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 distinguishes between the window of the terminal emulator and screens in the host application. For the purposes of testing, the terminal emulator window consists of the frame and menus of the terminal emulator itself. The terminal emulator 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 to test terminal emulator applications. It is recommended that you review the *WinRunner User's Guide* before you read this guide. If you are performing Year 2000 testing, also read the *Testing for Year 2000* guide.

Configuring Terminal Emulator Settings

You configure the terminal emulator settings when you install WinRunner. If, however, you need to modify any of the settings, select **Programs** > **WinRunner** > **Terminal Emulator Configuration** on the **Start** menu.

Creating Test Scripts

A test script consists of statements coded in Mercury Interactive's Test Script Language (TSL). These statements are generated automatically when you record, in response to input to the application. You can program statements manually, or mix recorded and programmed statements in the same test script.

By default, WinRunner records in Context Sensitive mode, meaning that the script reflects 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 learns these properties and uses them to identify and locate objects during a test run. For more information, see Chapter 2, **Context Sensitive Testing**.

The following is a sample of a WinRunner test script recorded on a terminal emulator application. 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");
```

For information on TSL functions, refer to the *TSL Online Reference* (**Help** > **TSL Online**).

Synchronizing Test Execution

When you record a test script, WinRunner inserts synchronization points automatically so that during a test run, execution will be delayed until the application is ready to receive input. You can also add synchronization points manually. For more information, see Chapter 3, **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 appearing when you run the test.

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

- **GUI Checkpoints**

  GUI checkpoints compare information about the screens and fields in your application interface *disregarding their location on screen*. You can add a GUI checkpoint that checks a single object, two or more objects, or an entire screen. For more information, see Chapter 4, **Checking Screens and Fields**.

- **Text Checkpoints**

  Text checkpoints compare on-screen text *according to its physical location on the screen*. WinRunner can capture the entire screen of the active terminal emulator window, or only the portion of the screen that you specify. For more information, see Chapter 5, **Checking Text**.

Testing VT100 and Text Applications

You can use WinRunner to test terminal emulator applications that do not support the EHLLAPI protocol. This includes terminal applications such as VT100, VAX, UNIX, HP, and text applications. For more information, see Chapter 6, **Testing VT100 and Text Applications**.

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. For more information, see Chapter 7, **Analyzing Results**.

Learning the Application with BMS Files

Before you can begin Context Sensitive testing, WinRunner 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, see Chapter 8, **Learning the Application with BMS Files**.

Using Default Command Softkeys

Some WinRunner commands can be activated 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.

The following tables show the softkey configurations available for RUMBA, Extra!, and NetSoft Elite. All other terminal emulator applications use the RUMBA default.

WinRunner Terminal Emulator Softkeys

The following table shows the softkeys available for testing a terminal emulator application.

Command	Softkey for 3270 (RUMBA)	Softkey for 5250 (RUMBA)	Softkey for VT100 (RUMBA)	Softkey for 3270 and 5250 (Extra!)	Softkey for 3270 and 5250 (NetSoft)
CHECK PARTIAL TEXT	PgDown	Left Ctrl + F3	Left Ctrl + F3	Left Ctrl + F4	Right Alt + PgUp
CHECK TEXT	Left Alt + PgUp	Left Ctrl + F1	Left Ctrl + F1	Left Ctrl + F2	Left Alt + F1

Command	Softkey for 3270 (RUMBA)	Softkey for 5250 (RUMBA)	Softkey for VT100 (RUMBA)	Softkey for 3270 and 5250 (Extra!)	Softkey for 3270 and 5250 (NetSoft)
CHECK PARTIAL DATE	Left Alt + End	Left Ctrl + F8	Left Ctrl + F8	Left Ctrl + F9	Left Alt + PgUp
CHECK DATE	Left Ctrl + PgDown	Left Ctrl + F2	Left Ctrl + F2	Left Ctrl + F3	Right Alt + PgDown
GET TEXT	Left Ctrl + End	Left Ctrl + F5	Left Ctrl + F5	Left Ctrl + F6	Left Alt + F4
EXCLUDE FILTER	Left Alt + PgDown	Right Ctrl + F7	Right Ctrl + F7	Left Ctrl + F7	Left Alt + F6
INCLUDE FILTER	Right Alt + PgDown	Left Ctrl + F7	Left Ctrl + F7	Left Ctrl + F8	Left Alt + F7
WAIT STRING	Right Ctrl + End	Left Ctrl + F12	Left Ctrl + F12	Left Ctrl + F5	Left Alt + End

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 with support for terminal emulator applications.

Command	Softkey for 3270 (RUMBA)	Softkey for 5250 (RUMBA)	Softkey for VT100 (RUMBA)	Softkey for 3270 and 5250 (Extra!)	Softkey for 3270 and 5250 (NetSoft)
RUN FROM ARROW	Left Ctrl + 7	Left Ctrl + 7	Left Ctrl + 7	Left Ctrl + 7	Right Alt + 7
RUN FROM TOP	Left Ctrl + 5	Left Ctrl + 5	Left Ctrl + 5	Left Ctrl + 5	Right Alt + 5
STEP	Left Ctrl + 6	Left Ctrl + 6	Left Ctrl + 6	Left Ctrl + 6	Right Alt + 6
STEP INTO	Left Ctrl + 8	Left Ctrl + 8	Left Ctrl + 8	Left Ctrl + 8	Right Alt + 8
STOP	Left Ctrl + 3	Left Ctrl + 3	Left Ctrl + 3	Left Ctrl + 3	Right Alt + 3
PAUSE	Pause	Pause	Pause	Pause	Pause

Command	Softkey for 3270 (RUMBA)	Softkey for 5250 (RUMBA)	Softkey for VT100 (RUMBA)	Softkey for 3270 and 5250 (Extra!)	Softkey for 3270 and 5250 (NetSoft)
STEP TO CURSOR	Left Ctrl + F9	Left Ctrl + F9	Left Ctrl + F9	Left Alt + 9	Left Alt + F10
RECORD	Scroll Lock	Left Alt + 2	Scroll Lock	Scroll Lock	Scroll Lock
MARK LOCATOR	Right Ctrl + 6	Right Ctrl + 6	Right Ctrl + 6	Left Alt + 6	Left Alt + 6
WAIT BITMAP	Left Ctrl + 0	Left Ctrl + 0	Left Ctrl + 0	Left Ctrl + 0	Right Alt + 0
CHECK BITMAP	Left Ctrl + PgUp	Right Ctrl + 0	Right Ctrl + 0	Left Ctrl + PageUp	Left Alt + F11
CHECK GUI	Right Ctrl + 2	Right Ctrl + 2	Right Ctrl + 2	Right Ctrl + 2	Left Alt + 2
GUI CHECK LIST	Right Ctrl + F12	Right Ctrl + F12	Right Ctrl + F12	Right Ctrl + F12	Right Alt + End

Command	Softkey for 3270 (RUMBA)	Softkey for 5250 (RUMBA)	Softkey for VT100 (RUMBA)	Softkey for 3270 and 5250 (Extra!)	Softkey for 3270 and 5250 (NetSoft)
WAIT BITMAP AREA	Left Ctrl + 4	Left Ctrl + 4	Left Ctrl + 4	Left Ctrl + 4	Right Alt + 4
CHECK BITMAP AREA	Left Ctrl + 2	Left Ctrl + 2	Left Ctrl + 2	Left Ctrl + 2	Right Alt + 2
GET TEXT AREA	Left Ctrl+1	Left Ctrl+1	Left Ctrl+1	Left Ctrl+1	Right Alt + 1
GET TEXT OBJECT	Left Ctrl + 9	Left Ctrl + 9	Left Ctrl + 9	Left Ctrl + 9	Right Alt + 9
INSERT FUNCTION FROM LIST	Left Alt + 7	Left Alt + 7	Left Alt + 7	Left Alt + 7	Left Alt + 7
INSERT FUNCTION	Left Alt + 8	Left Alt + 8	Left Alt + 8	Left Alt + 8	Left Alt + 8

Command	Softkey for 3270 (RUMBA)	Softkey for 5250 (RUMBA)	Softkey for VT100 (RUMBA)	Softkey for 3270 and 5250 (Extra!)	Softkey for 3270 and 5250 (NetSoft)
WAIT WINDOW	Left Alt + 9	Left Alt + 9	Left Alt + 9	Left Alt + F9	Left Alt + F9
GET TEXT	Left Ctrl + F10	Left Ctrl + F10	Left Ctrl + F10	Left Ctrl + F10	Right Alt + F10

Softkey assignments are configurable. If the application you are testing uses one of the default softkeys preconfigured for WinRunner, you can redefine the softkey by using the Softkey Configuration utility. Select **Programs** > **WinRunner** > **Softkey Configuration** on the **Start** menu. For more details, refer to the *WinRunner User's Guide.*

Sample Application

WinRunner includes a sample application you can use to experiment with testing terminal emulator applications. To start the Flight Reservation system, select **Programs** > **WinRunner** > **Sample Applications** > **Year2000 Demo Server** on the **Start** menu. For more information, refer to the **Year2000 Demo Server Read Me**.

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.

Context Sensitive Testing

You can use WinRunner's Context Sensitive features to test your terminal emulators applications. For general information on Context Sensitive testing with WinRunner, refer to the section "Understanding the GUI Map", in the *WinRunner User's Guide*.

This chapter describes:

- **Physical Descriptions**
- **Logical Names**
- **Object Classes for Terminal Emulators**
- **Properties**
- **Changing the Way Operations are Recorded**

About Context Sensitive Testing

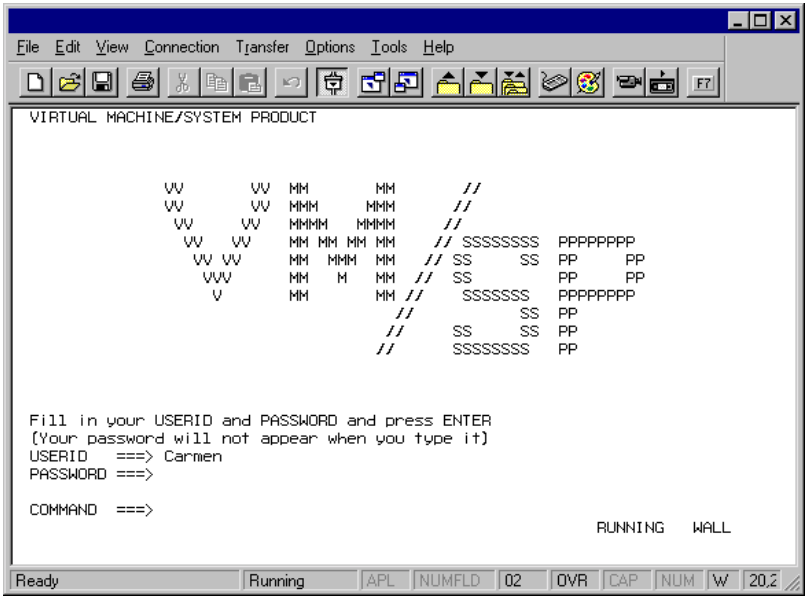
Context Sensitive testing ensures that non-essential changes in your application do not affect test execution. WinRunner can handle changes in window size between testing sessions, or modifications in the positioning of fields in an application screen. WinRunner records your operations in terms of the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

To perform Context Sensitive testing, WinRunner must uniquely identify each screen and field and be able to locate it in the application under test.

During Context Sensitive testing, WinRunner learns an accurate description of each object as it is identified by the terminal emulator application. If you have access to the BMS files of your application, WinRunner can learn your application by reading these files directly. See Chapter 8, **Learning the Application with BMS Files** for more information. Otherwise, WinRunner learns a description of each object using the 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 screen or field (called the *physical description)* contains a detailed list of properties. WinRunner places this list in a GUI map file. In the test script, WinRunner uses a *logical* name for each screen or field 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 learns the actual description, or list of properties, of both the screen and field 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 property 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 terminal emulator application.

Physical Descriptions

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

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

For example, the description of the “Login” screen presented above contains three properties, 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* property. 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 properties.

For more information on properties that are unique to WinRunner for terminal emulators, see **Properties** on page 25. For information on other properties used by WinRunner, refer to 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

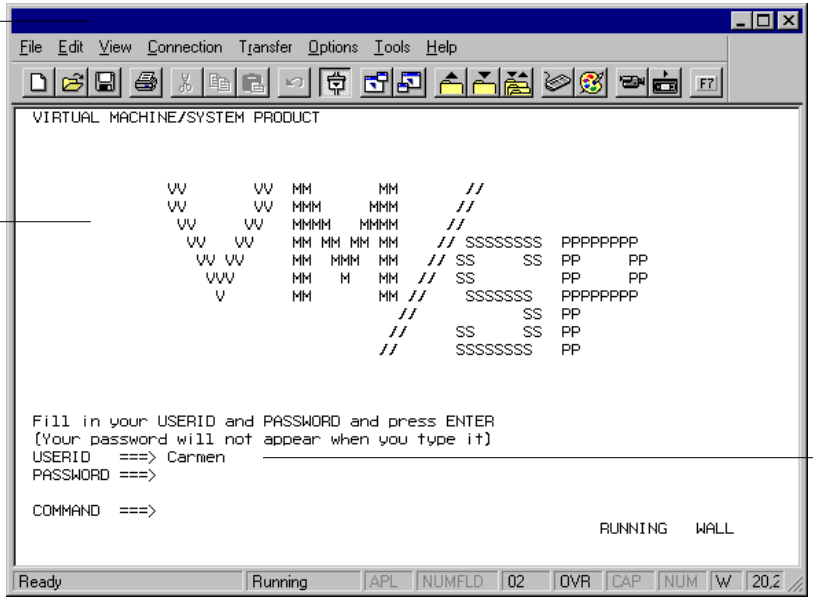
The logical name is the name WinRunner uses for objects in the test script. Once the name is assigned, you can modify it in the GUI map file.

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* property. The logical name of a field is the value of its *attached_text* property.

Object Classes for Terminal Emulators

WinRunner with add-in support for terminal emulators 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 also identifies the window of the terminal emulator, the outer frame of the terminal emulator including its menus and buttons. The class property of this window is always mic_if_window. For more information on this class, refer to the *WinRunner User's Guide.*



Properties

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

Screens

A screen can have the following properties:

Property	Description
class	The prime property 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 property that WinRunner uses. The value of this property is always 1.

Fields

A field can have the following properties:

Property	Description
class	The prime property 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.
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 screen origin.
y	The y coordinate of the top left corner of a field, relative to the screen 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 Operations are Recorded

When working with 3270 and 5250 protocol terminal emulators that support the EHLLAPI protocol, WinRunner records operations using the *field* or *position* method. The *field* method (default), enables WinRunner to record screens, fields, and PF keys using functions such as **TE_edit_field** and **TE_send_key**.

When the *position* method is used, WinRunner records keyboard and mouse input only. The operations on objects in your application are recorded as **win_type, obj_type**, **win_mouse_click**, and **win_mouse_drag** statements.

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

The current record method remains valid until you change it, even after you exit WinRunner and start it again. For more information on TSL, refer to the *TSL Online Reference*.

Synchronizing Test Execution

WinRunner 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**
- **Synchronizing Screen Changes**

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 depending on the system load. The screen refresh rate of your terminal emulator can also vary. WinRunner provides different types of synchronization points to pace test execution with the system. These points can be inserted into the test script automatically, using a softkey, or by programming.

Waiting for a Response from the Host

During recording, WinRunner 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.

Note: The **TE_wait_sync** function is only available for 3270 and 5250 terminal emulators that support the EHLLAPI protocol.

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 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 following example 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 for on the screen; 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. 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.

Note: The **TE_wait_field** function is only available for 3270 and 5250 terminal emulators that support the EHLLAPI protocol.

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.

Note: The **TE_set_refresh_time** function is only available for 3270 and 5250 terminal emulators that support the EHLLAPI protocol.

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 whether synchronization has been achieved before continuing test execution.

This statement has the following syntax:

TE_set_sync_time (*time* **);**

Note: The **TE_set_sync_time** function is only available for 3270 and 5250 terminal emulators that support the EHLLAPI protocol.

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* **);**

Note: The **TE_get_sync_time** function is only available for 3270 and 5250 terminal emulators that support the EHLLAPI protocol.

Synchronizing Screen Changes

In some AS/400 applications, you might have a case where typing a key in a specific field causes the screen to change. In such a case, WinRunner does not recognize that the screen has changed and does not generate the **TE_wait_sync** function.

In such cases, you can use the **TE_force_send_key** function and place it in your startup test. This function causes WinRunner to recognize that the screen has changed and to automatically generate **TE_wait_sync**. This function has the following syntax:

TE_force_send_key (*in_screen*, *in_field*, [*in_key*] **);**

The *in_screen* parameter defines the screen in which the field exists. The *in_field* parameter defines the field. The *in_key* parameter defines the input key which causes the screen to change (optional). You can use a key mnemonic (such as @E for Enter) or the WinRunner macros (such as TE_Enter for Enter).

The **TE_reset_all_force_send_key** resets the execution of the **TE_force_send_key** function. For more information about these functions, refer to the *TSL Online Reference*.

Checking Screens and Fields

WinRunner 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 a Single Field or a Screen**
- **Checking Two or More Fields**
- **Checking All Fields in a Screen at Once**
- **Properties for Screens and Fields**

About Checking Screens and Fields

GUI checkpoints allow you to check screens and fields in your application. For example, you can check the number of protected or input fields in a screen. Or you can check the content of a specific field, whether it is protected or visible.

To create a GUI checkpoint, you point to a screen or field and define the checks you want to perform. Information about the screens and fields as well as the checks is saved in a checklist. WinRunner captures the current state of these screens and fields 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. 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 Test Results window.

The information in this chapter applies specifically to GUI checks on terminal emulator applications. For additional information about GUI checkpoints, refer to the *WinRunner User's Guide.*

Checking a Single Field or a Screen

You can check a single field or screen by pointing at it and specifying the type of checks you want to perform.

To check a single field or a screen:

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

2 Double-click on the field or screen you want to check. (To check the screen, double-click on an empty area of the screen.)

Note: You can perform WinRunner's default checks by clicking once on the field or screen. The default check for a field is Date. The default checks for a screen are 'Number of protected fields' and 'Number of input fields'.

---

3 Select the checks you want to perform from the Check GUI dialog box. For more information, see **Properties for Screens and Fields** on page 44.

4 Click **OK**. WinRunner captures the screen or field information, stores it in the test's expected results folder, and inserts a **obj_check_gui** or a **win_check_gui** statement in your test script.

Checking Two or More Fields

You check two or more fields by creating a checklist while clicking on the fields you want to check.

To check two or more fields in a screen:

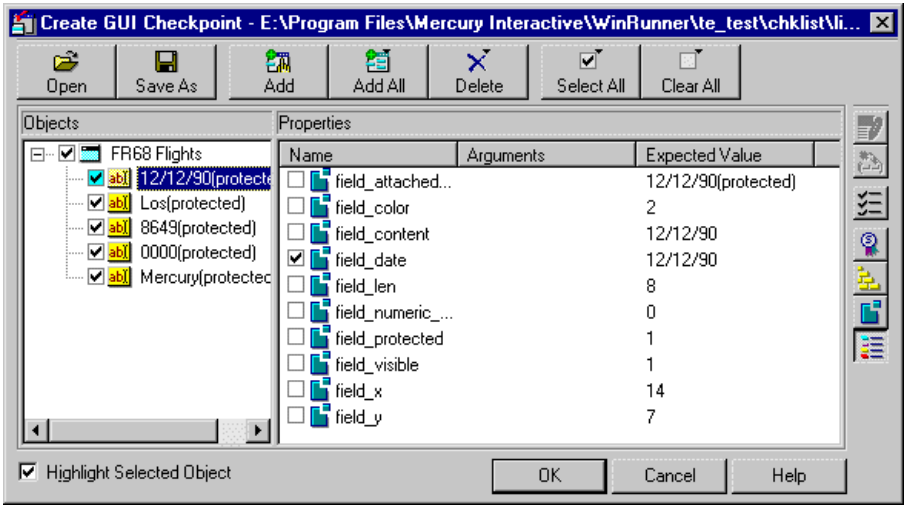
1 Choose **Create > GUI Checkpoint** > **For Multiple Objects**, or click the **GUI Checkpoint for Multiple Objects** button on the User toolbar. The Create GUI Checkpoint window opens.

2 Click the **Add** button.

3 Click once on each field you want to check.

4 Click the right mouse button to stop the selection process. The Create GUI Checkpoint window opens.

The **Objects** column lists the name of the screen and the fields you checked. The **Properties** column lists the properties for the selected field.



5 To modify a check, select the field in the **Objects** column and select the properties to be checked in the **Properties** column.

6 To save the checklist and perform the checks, click **OK**. WinRunner captures the information about the fields and stores it in the expected results folder. A **win_check_gui** statement is inserted in the test script.

Books Online
Find
Find Again
Help
Top of Chapter
Back

Checking All Fields in a Screen at Once

You can check all fields in a screen at once. WinRunner creates a checklist containing the default check ('Date') for all fields in the screen.

To check all the fields in a screen:

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

2 Click once on an empty area of the screen.

The Add All dialog box opens.

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

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

WinRunner captures the information about the fields and stores it in the test's expected results folder. (This may take several seconds.) The WinRunner window is restored and a **win_check_gui** statement is inserted into the test script.

Properties for Screens and Fields

When you create a GUI checkpoint, you can determine the types of checks to perform on screens and fields in your application.

Screen Checks

For a screen you can check the following properties:

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

For a field you can check the following properties:

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

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.

Date: checks the date of the field (default check)

Checking Text

You can use WinRunner to check the text in the screen of your terminal emulator application.

This chapter describes:

- **Checking Text Automatically**
- **Checking Text Using Softkeys**
- **Using Filters when Checking Text**
- **Reading Text from the Screen**
- **Searching for Text**

About Checking Text

WinRunner 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 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.

Checking Text Automatically

You can instruct WinRunner to automatically 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 a full screen automatic text check, execute the following statement in your test script:

TE_set_auto_verify (ON **);**

To deactivate automatic full screen text check, execute the following statement:

TE_set_auto_verify (OFF **);**

Each time a new full screen 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"* **);**

Note: The **TE_set_auto_verify** function is only available for 3270 and 5250 terminal emulators that support the EHLLAPI protocol.

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 a 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 the 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);

Note: The **TE_set_auto_verify** function is only available for 3270 and 5250 terminal emulators that support the EHLLAPI protocol.

Checking Partial Screens Using Previous Coordinates

When you choose the first/last partial text option, the coordinates for the partial screen automatic text 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 text 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);

Note: The **TE_set_auto_verify** function is only available for 3270 and 5250 terminal emulators that support the EHLLAPI protocol.

Checking Text Using Softkeys

During recording, you can use softkeys to check text. You can check the entire contents of the terminal emulator screen or you can check a specific portion of the screen. All captured text is stored as ASCII text.

Checking a Full Screen

Use a full screen text check to capture the entire contents of the active terminal emulator screen.

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 generated in your test script.

The entire contents of the active terminal emulator screen 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 folder 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 folder). 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 folder.

Checking a Partial Screen

Use a 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 the *TSL Online Reference*.

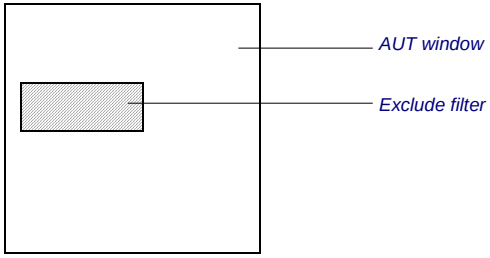
Using Filters when Checking Text

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*.

Note: You can also create filters to check dates on your screen.

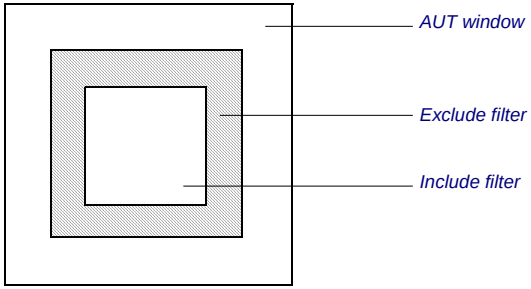
Exclude and Include Filters

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.

Creating Filters

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

To create a filter during 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 area.

3 To record the filter, click the right button.

WinRunner is restored. The filter is added to the test's db folder 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 all the screens in the terminal emulator application.

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

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 the *TSL Online Reference*.

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

Deactivating and Deleting Filters

When you deactivate an existing filter, it remains in the test's db folder 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, *screen_name* **);**

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 folder for later use. Use the **TE_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, *screen_name* **);**

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 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. Click the right mouse button to read the text. 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 the *TSL Online Reference*.

   Each new line of text that is captured is preceded by the characters “\n” in the variable. The following example shows how two lines of 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 the *TSL Online Reference*.

You can use WinRunner to test terminal emulator applications that do not support the EHLLAPI protocol. These include terminal emulator applications such as VT100, VAX, UNIX, HP, and text applications.

This chapter describes:

- **Creating Test Scripts**
- **Synchronizing Test Execution**
- **Checking Text**
- **TSL Functions**

About Testing VT100 and Text Applications

When working with VT100 terminal emulators, the *text* method is used. The text method is similar to the *position* method which can be used in 3270 and 5250 terminal emulators that support the EHLLAPI protocol. In both methods, WinRunner records keyboard and mouse input only. The operations on objects in your application are recorded as **win_type, obj_type**, **win_mouse_click**, and **win_mouse_drag** statements.

However, unlike the position method, the text method does not insert synchronization statements into your test script automatically. You need to insert synchronization statements using softkeys or by programming.

Creating Test Scripts

When using VT100 terminal emulator applications, WinRunner records keyboard and mouse input only. The operations on objects in your application are recorded as **win_type, obj_type**, **win_mouse_click**, and **win_mouse_drag** statements.

The following is a sample of a WinRunner test script recorded on a VT100 terminal emulator application. The comment (#) lines describe the statements.

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

# Direct input to the screen
set_window ("RUMBA - DEMO", 1);

# Type in the user id “Minnie”
obj_type ("AfxWnd40","minnie");

# Press the Enter key
obj_type ("AfxWnd40","<kReturn>");

# Wait for a string to appear on the next screen.
TE_wait_string(" MENU ", 1, 1, 53, 1, 60);

# Type a menu option.
obj_type ("AfxWnd40","90");
```

In the above example, the user clicks on the terminal emulator window to activate it. WinRunner records that action to ensure that the input is directed to the correct window. Then the user types the user name “Minnie” in the appropriate field and presses the Enter key.

A wait statement is added to ensure that WinRunner waits for the string to appear on the next screen. The user types an option.

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

Synchronizing Test Execution

When using VT100 terminal emulator applications, you can insert synchronization points to your test script in order to pace test execution with the system.

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. This function has the following syntax:

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

The *string* parameter lists 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.

For more information, see the "**Waiting for a Specific String**" section in Chapter 3, **Synchronizing 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 function has the following syntax:

TE_set_timeout (*timeout* **);**

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

Returning the Current Synchronization Time

The **TE_get_timeout** function returns the maximum time, in seconds, that WinRunner waits for response from the mainframe before continuing test execution. This function has the following syntax:

TE_get_timeout (*timeout* **);**

The default *timeout* is 60 seconds.

Setting Synchronization Keys

Using the **TE_define_sync_keys** function you can set keys that enable automatic synchronization in **type**, **win_type** and **obj_type** functions. When WinRunner executes a **type**, **win_type** or **obj_type** statement that includes a synchronization key, WinRunner waits for a specified string to either appear or disappear from the screen. This function has the following syntax:

TE_define_sync_keys (*keys*, *string*, *mode* [, *x1*, *y1*, *x2*, *y2*] **);**

The *keys* parameter is the keys that will enable synchronization. Use a comma as the delimiter between keys. The *string* parameter is the string that WinRunner waits for to appear or disappear on the screen. The *mode* parameter is one of the following: SYNC_WHILE (waits until the string disappears), SYNC_UNTIL (waits until the string appears), SYNC_DEFAULT (waits the default synchronization time). The parameters *x1, y1, x2, y2* define a rectangle in which to search for the string (optional). If these parameters are missing, WinRunner searches the entire screen.

Checking Text

While creating a test, you indicate the text that you want to check from a selected portion of the screen and store it in a file. The screen coordinates of the text you indicated are inserted into your test script.

Checking Text of a Terminal Emulator Screen

The **TE_check_text** function statement captures and compares the text in a terminal emulator window. This function has the following syntax:

TE_check_text (*file_name* **[** *,start_column, start_row, end_column, end_row*] **);**

The *file_name* parameter is a string expression given by WinRunner that identifies the captured window. The *start_column* and the *start_row* parameters are the column and row at which the captured text begins.

The *end_column* and *end_row* parameters are the column and row at which the captured text ends.

For more information, see the "**Checking Text Using Softkeys**" section in Chapter 5, **Checking Text**.

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.

This function has the following syntax:

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

The *string* parameter is the text that you want to locate. The *out_x_location* parameter is the output variable that stores the x coordinate of the test string. The *out_y_location* parameter is the output variable that stores the y coordinate of the text string. The *x1, y1, x2, y2* parameters describe a rectangle that defines the limits of the search area.

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

Reading Text from the Screen

Using the **TE_get_text** function, you can instruct WinRunner to read the text in a specified area of the screen and store it in a variable. This function has the following syntax:

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

The *x1, y1, x2, y2* parameters describe a rectangle that encloses the text to be read. The pairs of coordinates can designate any two diagonally opposite corners of the rectangle.

For more information, see the "**Reading Text from the Screen**" section in Chapter 5, **Checking Text**.

Using Filters

You can create 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. For more information, see the "**Using Filters when Checking Text**" section in Chapter 5, **Checking Text**.

TSL Functions

The following are TSL functions that you can use when testing your VT100 terminal emulator application. For more information on TSL functions, refer to the *TSL Online Reference*.

Synchronization Functions

You can insert synchronization points to your test script in order to pace test execution with the system.

- The **TE_define_sync_keys** function sets keys that enable automatic synchronization in **win_type** and **obj_type** commands. It has the following syntax:

  TE_define_sync_keys (*keys, string, mode* **[**, *x1, y1, x2, y2*] **);**

- The **TE_get_timeout** function returns the current synchronization time. It has the following syntax:

  TE_get_timeout ( );

- The **TE_set_timeout** function sets the maximum time WinRunner waits for a response from the server. It has the following syntax:

  TE_set_timeout (*timeout* **)**;

- The **TE_wait_string** function waits for a string to appear on screen. It has the following syntax:

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

Text Functions

You can check the text in the screen of your terminal emulator application.

- The **TE_check_text** function captures and compares the text in a terminal emulator window. It has the following syntax:

  TE_check_text (*file_name* **[** *,start_column, start_row, end_column, end_row* **]);**

- The **TE_find_text** function returns the location of a specified string. It has the following syntax:

  TE_find_text (*string, out_x_location, out_y_location* **[***x1, y1, x2, y2* **]);**

- The **TE_get_text** function reads text from screen and stores it in a string. It has the following syntax:

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

Filter Functions

You can create filters to check text and dates on your terminal emulator screen.

- The **TE_create_filter** function creates a filter in the test database. It has the following syntax:

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

- The **TE_delete_filter** deletes a specified filter from the test database. It has the following syntax:

  TE_delete_filter (*filter_name* **)**;

- The **TE_get_active_filter** function returns the coordinates of a specified active filter. It has the following syntax:

  TE_get_active_filter (*filter_num* **[***out_start_column, out_start_row, out_end_column, out_end_row***]**, *screen_name* **);**

- The **TE_get_auto_reset_filter** function indicates whether or not filters are automatically deactivated at the end of a test run. It has the following syntax:

  TE_get_auto_reset_filters ( );

- The **TE_get_filter** function returns the properties of a specified filter. It has the following syntax:

  TE_get_filter (*filter_name* **[**,*out_start_column, out_start_row, out_end_column, out_end_row, out_type, out_active, screen_name***]);**

- The **TE_reset_all_filters** function deactivates all filters in a test. It has the following syntax:

  TE_reset_all_filters ( );

- The **TE_reset_filter** function deactivates a specified filter. It has the following syntax:

  TE_reset_filter (*filter* **);**

- The **TE_set_filter** function creates and activates a filter. It has the following syntax:

  TE_set_filter (*filter_name* **[**,*start_column, start_row, end_column, end_row,* EXCLUDE | INCLUDE, *screen_name*] **);**

- The **TE_set_auto_reset_filters** function deactivates the automatic reset of filters when a test run is completed. It has the following syntax:

  TE_set_auto_reset_filters (ON | OFF **);**

- The **TE_set_filter_mode** function specifies whether to assign filters to all screens or to the current screen. It has the following syntax:

  TE_set_filter_mode (*mode* **);**

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.

This chapter describes:

- **Viewing Results of a GUI Checkpoint**
- **Viewing Results of a Text Checkpoint**

About Viewing Test Results

When a test run is completed, you can view detailed test results in the WinRunner Test Results window. To open the dialog box, select **Tools** > **Test Results** or click the **Test Results** button. The Test Results window opens and displays the results of the current test. You can view expected, debug, and verification results in the Test Results window. By default, the Test Results window displays the results of the most recently executed test run. For more information, refer to the *WinRunner User's Guide*.

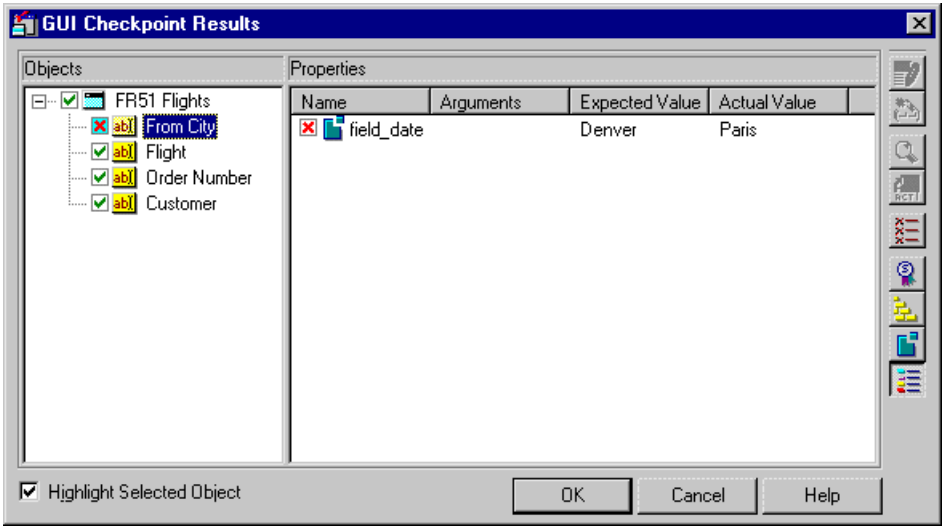
Viewing Results of a GUI Checkpoint

A GUI checkpoint compares expected and actual results in your application. You can view the expected and actual results through the Test Results window. If a mismatch is detected during a verification run, you can view the differences between the expected and actual results.

To view the results of a GUI checkpoint:

1 Open the Test Results window. In the test log, look for entries that list “end GUI checkpoint” in the **Event** column. Failed GUI checkpoints appear in red; passed GUI checkpoints appear in green.

2 Double-click an “end GUI checkpoint” entry in the log. Alternatively, highlight the entry and choose **Options > Display** or click the **Display** button. The GUI Checkpoint Results dialog box opens.



3 Click an object in the **Objects** column.

4 Select a property from the **Properties** column.

If the property is a field_date, click the **Compare expected and actual values** button. The Check Date Results dialog box opens.

Viewing Results of a Text Checkpoint

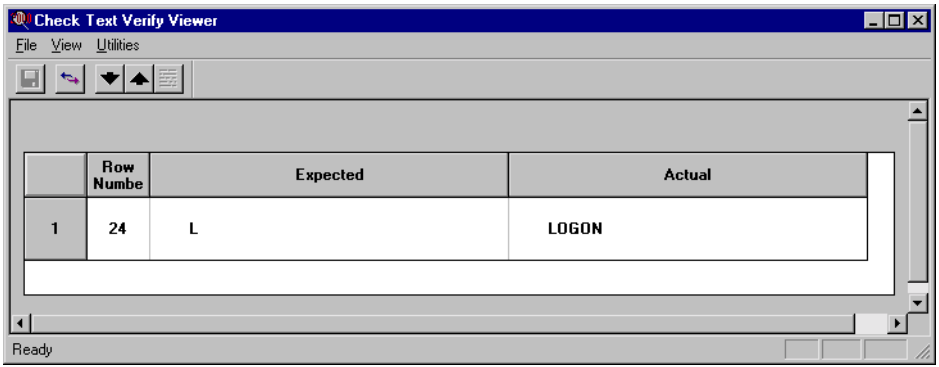
A text checkpoint compares expected and actual text in your application. You can view the expected and actual results using the Test Results window. If a mismatch is detected during a verification run, you can also view a file showing the differences between the expected and actual results.

To view the results of a text checkpoint:

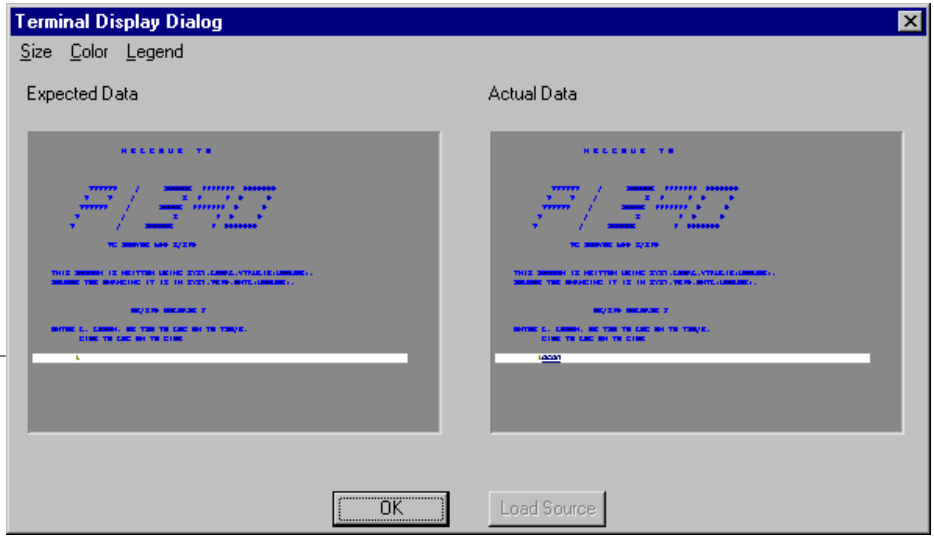
1 Open the Test Results window. In the test log, look for entries that list “check text” in the **Event** column. Failed text checkpoints appear in red; passed text checkpoints appear in green.

2 Double-click a “check text” entry in the log. Alternatively, highlight the entry and choose **Options > Display** or click the **Display** button.

For an entry with no mismatch, the Terminal Display Dialog opens. For an entry with a mismatch, the Check Text Verify Viewer opens.



3 To view the results in a mainframe view, double-click the row entry. The Terminal Display Dialog opens.



mismatch

4 Click **OK** to close the Terminal Display Dialog.

Books Online
Find
Find Again
Help
Top of Chapter
Back

Learning the Application with BMS Files

The Learn BMS Files feature can teach WinRunner 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 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 Chapter 2, **Context Sensitive Testing**, as well as the “Understanding the GUI Map” section in the *WinRunner User’s Guide* before you use the Learn BMS Files feature.

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 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. For more information on the GUI Map Editor, refer to the *WinRunner User's Guide*.

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);
```

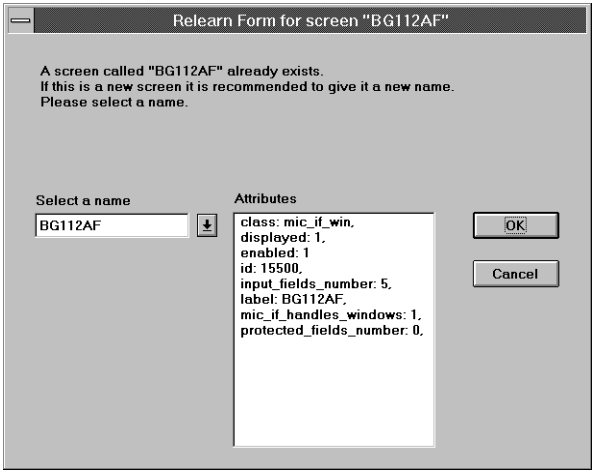
As WinRunner converts the BMS file into the GUI map file, it looks for discrepancies between the BMS file learned using the LEARN option and the current file, on which RELEARN is performed. Each time it finds a mismatch, a dialog box appears on screen and asks your how to proceed.

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.

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 properties. 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 properties of the selected object, according to the new BMS file. By default, WinRunner updates the GUI map to include the new properties.

Books Online
Find
Find Again
Help
Top of Chapter
Back

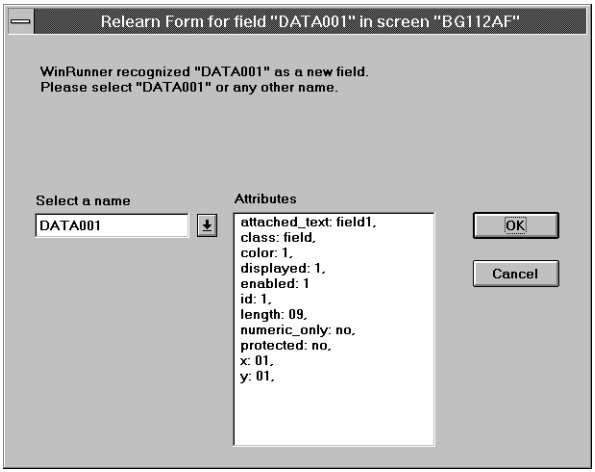
Click **OK**. The following message appears: “Screen BG112AF is now changed and gets new properties”. Click **OK**.

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.

Object Is Not in the Original GUI Map File



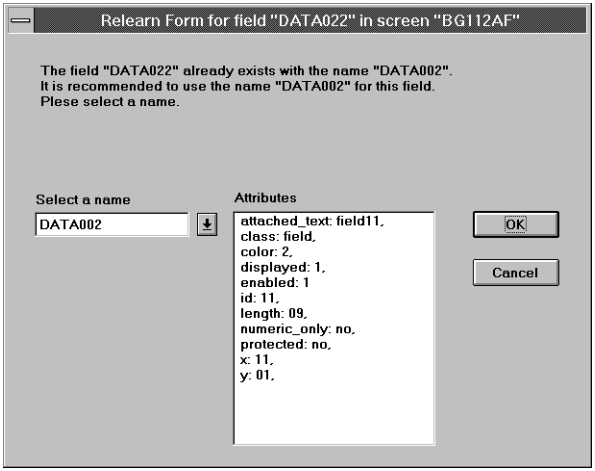
WinRunner found a field that it recognizes as a new one: no other field with the same name or properties exists in the GUI map file. The name of the field is displayed in the list on the left side of the Relearn dialog box. The list on the right shows all the properties of the selected field, according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

Object Appears in the GUI Map File with a Different Name



WinRunner found a field with the same properties 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.

Index

P

R

S

T

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

V

W

Y

WinRunner Testing Terminal Emulator Applications, Version 6.0

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

Mercury Interactive Corporation
1325 Borregas Avenue
Sunnyvale, CA 94089
Tel. (408) 822-5200 (800) TEST-911
Fax. (408) 822-5300

WRTEUG6.0/01