MERCURY
WINRUNNER™

Customization Guide

MERCURY™

Mercury WinRunner
Customization Guide
Version 7.6

MERCURY™

Mercury WinRunner Customization Guide, Version 7.6

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

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

Table of Contents

Welcome to WinRunner Customization

Welcome to WinRunner Customization. By customizing various aspects of WinRunner, you extend WinRunner's ability to meet your testing requirements.

Using This Guide

This guide describes the main concepts behind customizing WinRunner. It provides step-by-step instructions to help you extract the most out of WinRunner, while ensuring that you meet the testing requirements of your application.

This guide contains three parts:

Part I   Customizing GUI Checks

Describes how to develop and implement custom checks for custom GUI objects.

Part II   Customizing Recording

Describes how to customize the way in which WinRunner records operations on custom objects in order to improve test script readability.

Part III   Using the WinRunner API

Describes how to use WinRunner's API functions to enable you to test application functionality that is invisible at the level of the user interface, and applications that have no user interface.

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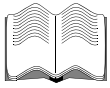
In addition to this guide, WinRunner comes with a complete set of documentation:

WinRunner User's Guide explains how to use WinRunner to meet the special testing requirements of your application.

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

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

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

Online Resources

In addition to this guide, WinRunner includes the following online resources:

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

Books Online displays the complete documentation set in PDF format. Online books can be read and printed using Adobe Acrobat Reader. It is recommended that you use version 5.0 or later. You can download Adobe Acrobat Reader from www.adobe.com. Check Mercury Interactive's Customer Support web site for updates to WinRunner online books.

WinRunner Context-Sensitive Help provides immediate answers to questions that arise as you work with WinRunner. It describes menu commands and dialog boxes, and shows you how to perform WinRunner tasks. Check Mercury Interactive's Customer Support web site for updates to WinRunner help files.

TSL Online Reference describes Test Script Language (TSL), the functions it contains, and examples of how to use the functions. Check Mercury Interactive's Customer Support site for updates to the *TSL Online Reference*.

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

Technical Support Online uses your default web browser to open Mercury Interactive's Customer Support web site.

Support Information presents Mercury Interactive's home page, its Customer Support web site, and a list of Mercury Interactive's offices around the world.

Mercury Interactive on the Web uses your default web browser to open Mercury Interactive's home page. This site provides you with the most up-to-date information on Mercury Interactive, its products and services. This includes new software releases, seminars and trade shows, customer support, training, and more.

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.
Arial	The Arial font is used for examples and statements that are to be typed in literally.
[ ]	Square brackets enclose optional parameters.
{ }	Curly brackets indicate that one of the enclosed values must be assigned to the current parameter.
...	In a line of syntax, 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 have been intentionally omitted.
|	A vertical bar indicates that either of the two options separated by the bar should be selected.

Part I

Customizing GUI Checks

1

Introduction

WinRunner offers an extensive array of features that you can use to test your software. You can extend these capabilities by customizing WinRunner to meet the particular requirements of your application. This guide provides detailed information on how you can customize WinRunner to enhance your testing capabilities.

You can customize WinRunner in the following areas:

➤ **GUI checks**

If WinRunner's standard GUI checks do not completely meet your specific testing requirements, you can extend your verification capabilities by creating custom property checks.

➤ **Recording test scripts**

If your application contains custom objects, you can ensure that your test scripts are easy to read and understand by customizing the functions that WinRunner records into the scripts.

➤ **Using the Mercury API**

You can use the Mercury API (Application Programming Interface) to record and execute functions in your application that are not connected to the user interface.

Customizing GUI Property Checks

WinRunner provides a variety of property checks for checking the GUI objects in your application. If WinRunner's standard property checks do not completely meet your testing requirements, you can extend your testing capabilities by creating custom GUI property checks. With WinRunner, you can customize GUI property checks in three ways:

➤ Chapter 2, "Creating Custom GUI Checks for Standard Objects," describes how to develop custom property checks to perform on standard GUI objects. For example, you can develop a property check for the size of the font used in an editor. When you associate the new property check with the standard edit class, it is displayed in the GUI checkpoint dialog boxes whenever you create or edit a checkpoint for an edit class object.

The following Check GUI dialog box displays the standard property checks for edit class objects:
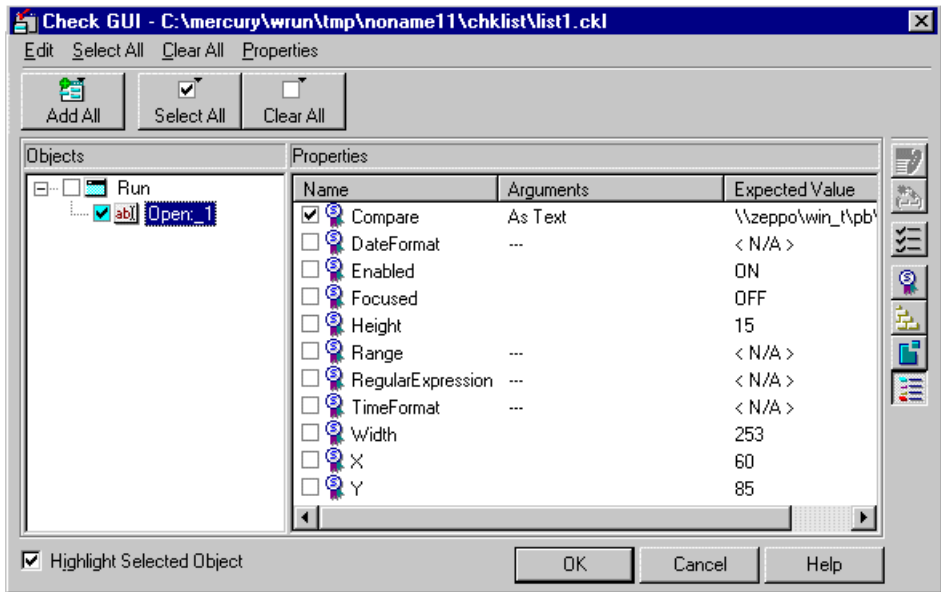
The following Check GUI dialog box displays a custom property check as well as the standard property checks for edit class objects:
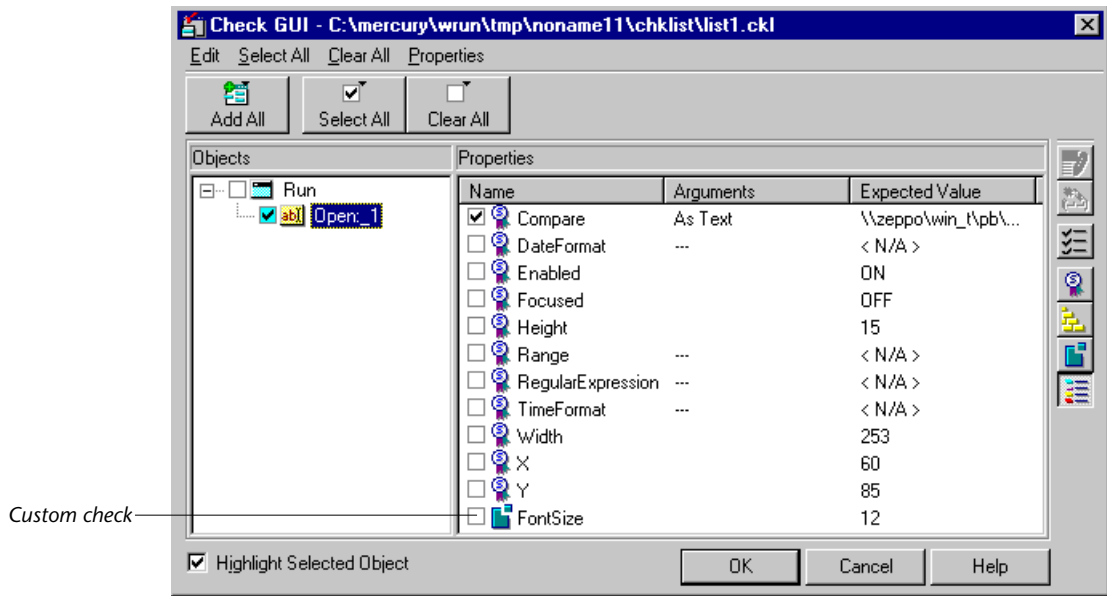
Custom check

➤ Chapter 3, "Creating GUI Checks for Custom Objects," describes what to do if your application has GUI objects that do not belong to any of WinRunner's standard classes. You can create custom verification classes for these objects, and then specify which property checks to include when checking objects of these custom classes.

For example, you can create a custom class for verification called "pbTool." When you create a GUI checkpoint on a pbTool class object, the available property checks are displayed in the Properties pane in either the Check GUI dialog box or the Create GUI Checkpoint dialog box. You can also add customized property checks for this new custom class. These customized property checks are displayed in the GUI checkpoint dialog boxes whenever you create or edit a GUI checkpoint on objects of that class.

The following Check GUI dialog box displays the standard property checks on an object belonging to a custom class. A custom object is any object that does not belong to one of the standard classes used by WinRunner.

These objects are assigned to the generic "object" class, which includes the following checks:
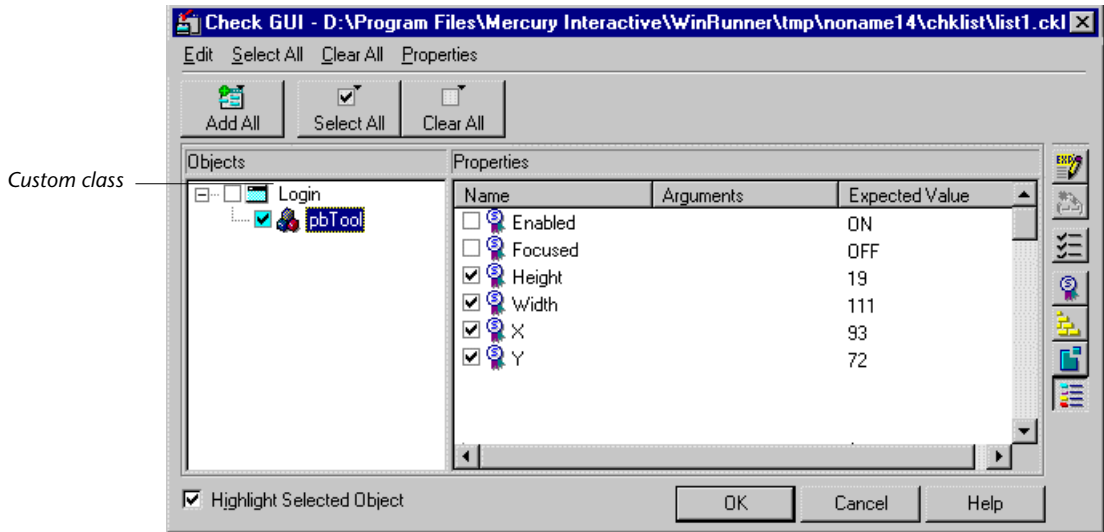
Custom class

You can also add custom checks to a custom class:

Custom class
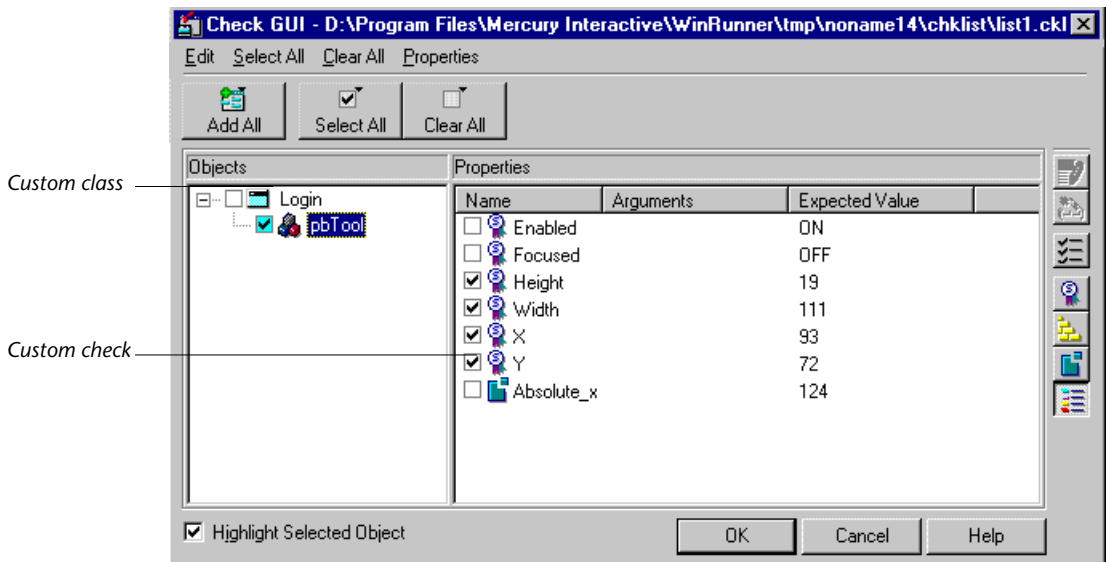
Custom check

➤ Chapter 4, "Creating GUI Checks: Advanced Topics," describes how to create your own user interface for the GUI checkpoint dialog box that is opened when you create a check on a GUI object belonging to a custom object class. The chapter also describes how to implement a custom utility to display the results of a custom check.

Customizing Recording

When you record operations on standard GUI objects, WinRunner creates test scripts that are easy to read and understand. However, when you operate on custom GUI objects, whose behavior differs significantly from that of WinRunner's standard objects, the resulting test script contains generic **obj_** TSL statements.

If your application contains custom objects, you can ensure that your test scripts are easy to read and understand by customizing the functions that WinRunner records into the scripts. WinRunner enables you to customize recorded statements in three ways:

➤ Chapter 5, "Customizing Recorded Statements," describes how to specify the function calls that WinRunner records into your test scripts when you operate on custom GUI objects.

➤ Chapter 6, "Adding Custom Properties for GUI Objects," describes how to add your own properties to any GUI object class to improve WinRunner's ability to uniquely identify the GUI objects in your application.

➤ Chapter 7, "Customizing Assigned Logical Names," describes how to customize the way that WinRunner assigns logical names to custom GUI objects in your application. By doing so, it is immediately apparent which recorded statement refers to which GUI object.

Using the WinRunner API

Chapter 8, "The Mercury API Functions," describes all the Mercury API functions that you need for inside testing.

2

Creating Custom GUI Checks for Standard Objects

You can enhance WinRunner's ability to check GUI objects in your application by developing custom property checks to perform on standard GUI objects.

This chapter describes:

➤ Creating a Capture Function

➤ Creating a Comparison Function

➤ Registering a New Property Check

➤ Associating a New Property Check with a GUI Object Class

➤ Modifying the Default Checks for a GUI Object Class

About Creating Custom GUI Checks for Standard Objects

By inserting a GUI checkpoint into a test script, you instruct WinRunner how and when to check the status of your application. As part of the process of inserting a checkpoint, you define which object properties WinRunner will check. For each GUI object class, WinRunner has a set of standard property checks from which you select. If the standard property checks do not fulfill your testing requirements, you can develop your own *custom* property checks. You add custom property checks to a GUI checkpoint dialog box for the class of objects you want to check. For information about standard checks on GUI objects and the GUI Checkpoint dialog boxes, refer to the "Checking GUI Objects" chapter in the *WinRunner User's Guide*.

For example, suppose you want to check the size of the font used in an edit box. You can develop a property check to check this property. If you associate the new check with the standard edit class, the edit object is highlighted in the Objects pane and the customized font size property is displayed along with the standard edit class properties in the Properties pane.

The Check GUI dialog box below displays the standard edit class property checks:
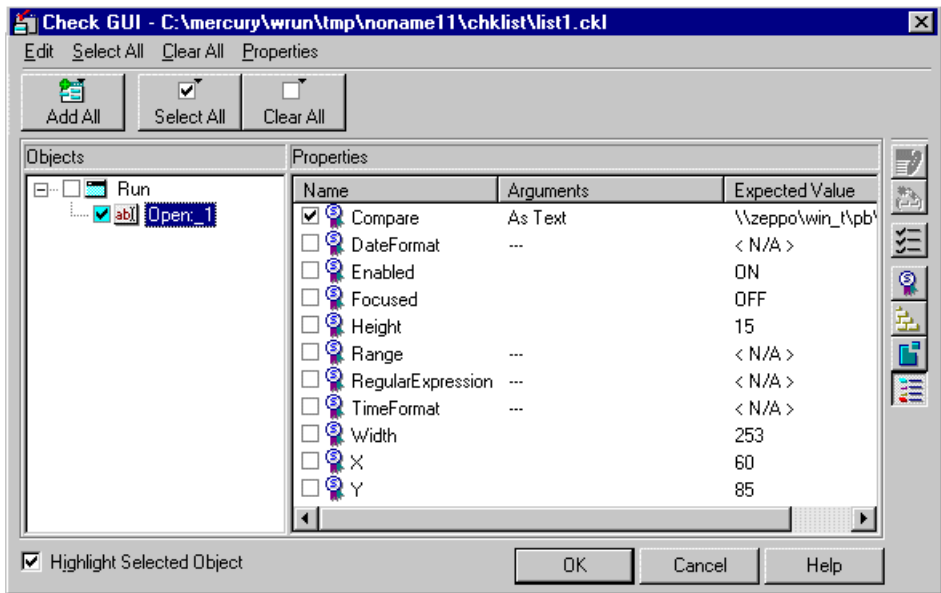
The Check GUI dialog box below displays both the custom font size property check and the standard edit class property checks:

Check GUI - C:\mercury\wrun\tmp\noname11\chklist\list1.ckl			

Edit Select All Clear All Properties

📇 Add All	☑ Select All	☐ Clear All		

Objects | **Properties**

□ ■ Run
　　☑ abl Open:_1

Name	Arguments	Expected Value
☑ 🔍 Compare	As Text	\\zeppo\win_t\pb\...
☐ 🔍 DateFormat	---	< N/A >
☐ 🔍 Enabled		ON
☐ 🔍 Focused		OFF
☐ 🔍 Height		15
☐ 🔍 Range	---	< N/A >
☐ 🔍 RegularExpression	---	< N/A >
☐ 🔍 TimeFormat	---	< N/A >
☐ 🔍 Width		253
☐ 🔍 X		60
☐ 🔍 Y		85
☐ 📘 FontSize		12

Custom check

☑ Highlight Selected Object　　　　　　　OK　　Cancel　　Help
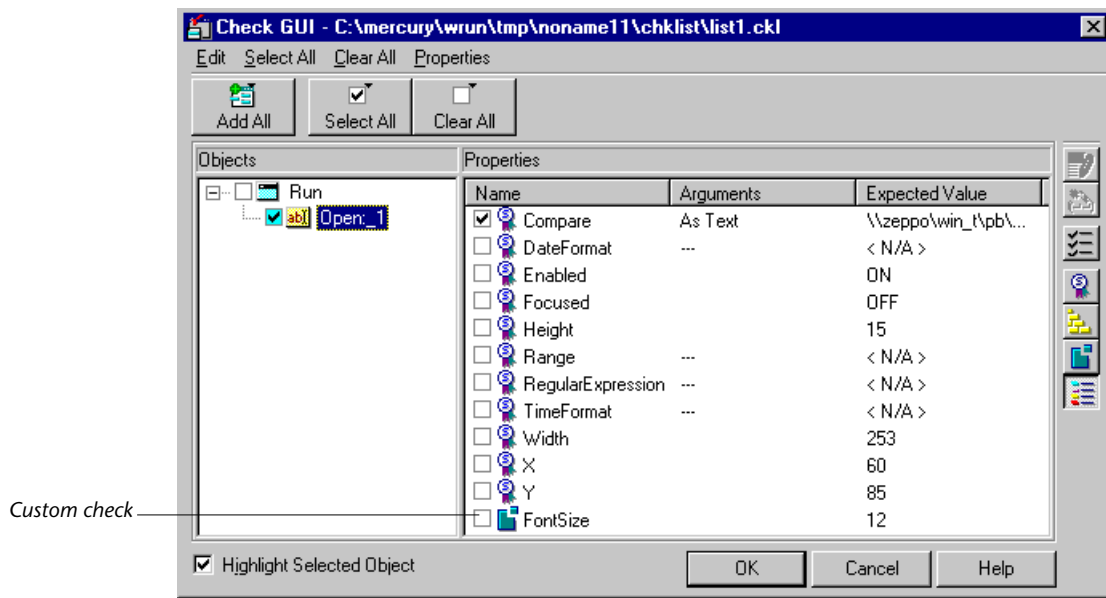
To add a custom property to a GUI Checkpoint dialog box for a standard WinRunner GUI object class, you perform the following steps:

1 Create a function to capture the expected and actual results of the custom property.

2 Create a function to compare the expected and the actual results.

3 Register the property.

4 Associate the property with a standard GUI object class.

5 Set the new property as a default property for the GUI object class (optional).

You can use WinRunner's Function Generator to generate all the required function calls, and then insert them directly into your test scripts. You can find the functions in the "GUI verification" category of the Function Generator. For more information on automatically generating and inserting functions, refer to the "Generating Functions" chapter in the *WinRunner User's Guide*.

Before using the capture and comparison functions, you must compile them. Although you can do this by running the functions from a test script, it is recommended that you include them in a compiled module and load the module from a startup test. This makes the functions available for all your WinRunner sessions. For more information on compiled modules, refer to the "Creating Compiled Modules" chapter in the *WinRunner User's Guide*.

You can add properties to the GUI Checkpoint dialog box for a standard WinRunner GUI object class, as described in this chapter. Alternatively, if your application has GUI objects that do not belong to any of WinRunner's standard classes, you can define a new custom GUI object class for verification, and then:

➤ specify which properties are available to the new class. For more information, see Chapter 3, "Creating GUI Checks for Custom Objects."

➤ customize the GUI checkpoint dialog boxes with a custom user interface and custom result display utility for the new class. For more information, see Chapter 4, "Creating GUI Checks: Advanced Topics."

Besides adding new property checks for a standard WinRunner class, you can also add to or change the properties that are checked by default for a class. For example, a standard check on a push button, by default, checks only that the push button is enabled. You can specify any other standard or custom properties that are checked by default for the push_button class. For instance, you might want to include the button's label as a default check.

Creating a Capture Function

You create a capture function to establish and store the expected and actual results of a custom check. For example, if you develop a check for the size of the font used in an edit box, it is the capture function that actually determines and stores the font size.

The capture function has the following syntax:

public **capture_function_name (** in *object*, inout *value* [, in *arg_list*] **)**

➤ *capture_function_name* is the name of your capture function.

➤ *object* is an *in* parameter that is assigned the description of the GUI object to check.

➤ *value* is an *inout* parameter:

  ➤ If the result of the capture function is a number or a string, then the capture function assigns the result of the function to the *value* parameter. For example, in the above example you would assign the font size, say "10", to the *value* parameter.

  ➤ If the result of the capture function is either very long (greater than 2Kb), or not in text format, then the capture function must store the result in a file. You use the unique filename that WinRunner assigns to the *value* parameter.

➤ *arg_list* is an optional *in* parameter that is passed from the *ui_function* parameter. You use the *ui_function* parameter only if you use the **gui_ver_add_class** function to creating a GUI checkpoint dialog box with a custom user interface. For more information, see Chapter 4, "Creating GUI Checks: Advanced Topics."

To make your capture function available to all tests, declare the function as *public*. You replace *capture_function_name* with the name of your capture function. For more information on user-defined functions, refer to the "Creating User-Defined Functions" chapter in the *WinRunner User's Guide*.

Example 1: Checking the Absolute X-Coordinate of an Object

In the following example, the user-defined *get_abs_x* capture function returns the x-coordinate of the top left corner of a GUI object, relative to the screen origin. The **obj_get_info** TSL function is called to determine the object's screen coordinate, abs_x.

```
public function get_abs_x (in object, inout value)
{
    return (obj_get_info (object, "abs_x", value));
}
```

The following example presents a mechanism for storing the results of a capture function in a file.

```
public function get_abs_x(in object, inout file)
{
    auto x_coord, rc;
    rc=obj_get_info(object, "abs_x", x_coord);
    file_open(file,FO_MODE_WRITE);
    file_printf(file,"%s",x_coord);
    file_close(file);
    return(rc);
}
```

Example 2: Checking Text Color

The following example verifies the color of the text in an edit field. The *edit_get_text_color* capture function uses the Windows API function *GetDC* to get the device context of the edit field. The *GetPixel* function is used, first to find the background color of the edit box, and then to find the foreground color. For more information on using API functions, refer to the "Calling Functions from an External Library" chapter in the *WinRunner User's Guide*.

```
# load Windows API declarations
load("c:\\wrun\\lib\\win_api",1,1);

# Capture Function
public function edit_get_text_color(in obj_name, inout rgb_val)
{
    auto hWnd, hDc, ret, i, w, h, backcolor, rc;
```

```
# get edit field's width and height
rc=obj_get_info(obj_name, "handle", hWnd);
if(rc != E_OK)
    return(rc);
rc=obj_get_info(obj_name,"height",h);
if(rc != E_OK)
    return(rc);
rc=obj_get_info(obj_name,"width",w);
if(rc != E_OK)
    return(rc);

# get edit field's device context
hDc=GetDC(hWnd);

# find background color
backcolor=GetPixel(hDc,2,2);

# find foreground color by scanning the edit field
for (i=1; i<w;i++)
{       ret=GetPixel(hDc,int(h/2),i);
    if((ret != backcolor)|| (ret==0)) break;
}

# release device context
ReleaseDC(hWnd, hDc);
rgb_val=ret;
return(E_OK);
}
```

Creating a Comparison Function

After the capture function has determined the expected and actual results for a custom check, WinRunner verifies the results to determine whether the check passed. To verify a check, you can either use WinRunner's standard comparison function, **default_compare_func**, or create your own comparison function.

Using the Standard Comparison Function

The **default_compare_func** function compares the expected results to the actual results. If the results match, the **default_compare_func** function returns 0, otherwise the function returns 1. The function compares the results according to their format—that is, either as a number or as a string.

The **default_compare_func** function works in all cases where there is a simple comparison between expected and actual results. For example, you would use the **default_compare_func** function if you were checking the absolute x coordinate of an object. If the expected coordinate was 250, and the actual coordinate was 200, then the **default_compare_func** function would return 1, indicating a mismatch. If the expected and actual coordinates were both 250, then the **default_compare_func** function would return 0, indicating a successful check.

Creating Your Own Comparison Function

If a complicated comparison is required to determine the success of a check, then you must develop your own comparison function. For example, suppose you are comparing dates that can have different formats, such as 8 March 1996 or 08/03/1996. Because the **default_compare_func** function is ineffective in this case, you must develop your own comparison function.

The comparison function has the following syntax:

```
public comparison_function_name ( in expected, in actual [, in arg_list] [, inout
display_information] )
{
    ...
    return(Mercury_error_code);
}
```

➤ *comparison_function_name* is the name of your comparison function.

➤ *expected* is an *in* parameter. If the capture function assigned the expected results to the *value* parameter, then the *expected* parameter is assigned the value of the expected results. If the capture function stored its results in a file, then the *expected* parameter is assigned the path and file name of the results file.

➤ *actual* is an *in* parameter. If the capture function assigned the actual results to the *value* parameter, then the *actual* parameter is assigned the value of the actual results. If the capture function stored its results in a file, then the *actual* parameter is assigned the path and file name of the results file.

➤ *arg_list* is an optional *in* parameter that is passed from the *ui_function* parameter. You use the *ui_function* parameter only if you use the **gui_ver_add_class** function to creating a GUI checkpoint dialog box with a custom user interface. For more information, see Chapter 4, "Creating GUI Checks: Advanced Topics."

➤ *display_information* is an optional *inout* parameter that is used only if you specify your own display function for the results of the check. WinRunner assigns to the *display_information* parameter a unique file name. You can store in this file information that you want to use when you call your display function. For more information, see Chapter 4, "Creating GUI Checks: Advanced Topics."

To make your comparison function available to all tests, declare the function as *public*. You replace *comparison_function_name* with the name of your comparison function. For more information on user-defined functions, refer to the "Creating User-Defined Functions" chapter in the *WinRunner User's Guide*.

The comparison function must return a Mercury error code: E_OK when the expected and actual results match; E_DIFF when the results do not match. E_DIFF is the error code for "GUI verification mismatch found."

Example 1: Simple Comparison Function

In the following example, a user-defined comparison function called
compare_number checks whether the expected and actual results of the
check are the same. An if/else statement is used to return E_OK if the results
match, and E_DIFF if they do not. Note that you could use the
default_compare_func function in this scenario.

```
public function compare_number(in expected, in actual)
{
    if (expected==actual)
        return(E_OK);
    else
        return(E_DIFF);
}
```

Example 2: Retrieving Results from a File

This example assumes that the capture function stored the expected and
actual results in files. The file names are passed to the comparison function
as exp_file and act_file respectively.

```
public function compare_number(in exp_file, in act_file)
{
    auto expected, actual;

    file_open(exp_file,FO_MODE_READ);
    file_getline(exp_file, expected);
    file_close(exp_file);
    file_open(act_file,FO_MODE_READ);
    file_getline(act_file, actual);
    file_close(act_file);
    if (expected==actual)
        return(0);
    else
        return(E_DIFF);
}
```

Registering a New Property Check

Once you have created and compiled the capture and comparison functions, you must register the new check. This is done using the **gui_ver_add_check** TSL function.

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

gui_ver_add_check (*check_name, capture_function, comparison_function* [,*display_function*] [,*type*] **);**

➤ *check_name* defines the name of the check. Note that the name of the check cannot contain spaces. The *check_name* will appear at the bottom of the appropriate check dialog box. See "Associating a New Property Check with a GUI Object Class" on page 20.

➤ *capture_function* is the name of the capture function that you developed for the check.

➤ *comparison_function* is either the **default_compare_func** function, or the name of the comparison function that you developed for the check.

➤ *display_function* is an optional parameter that enables you to use your own display utility to view the results of a check. You use the *display_function* parameter only if you use the **gui_ver_add_class** function to creating a GUI checkpoint dialog box with a custom user interface. For more information, see Chapter 4, "Creating GUI Checks: Advanced Topics."

➤ *type* is an optional parameter that indicates whether the check is for a window (1) or for any other GUI object (0). If no *type* is declared, the default (0) is assumed.

In the following example, the **gui_ver_add_check** function registers a check for an object's absolute x-coordinate.

gui_ver_add_check("Absolute_x","get_abs_x","compare_number", "",0);

The following example shows how to specify the **default_compare_func** function when you register a check.

gui_ver_add_check("Absolute_x","get_abs_x","default_compare_func", "",0);

For additional information about the **gui_ver_add_check** function, refer to the *TSL Online Reference*.

Associating a New Property Check with a GUI Object Class

Once you have registered the new property check, you must associate it with a GUI object class. By associating the new property check with a class, you add the property check to the bottom of the list of properties displayed for that class in the GUI checkpoint dialog boxes.

You associate a property check with a class using the **gui_ver_add_check_to_class** TSL function. This function has the following syntax:

gui_ver_add_check_to_class (*class*, *property_check_name* **);**

➤ *class* is the name of either the MSW_class or the standard class with which the check is associated.

➤ *property_check_name* is the name of the property check you defined using the **gui_ver_add_check** TSL function. The new property check will appear at the bottom of the list of properties displayed for the class in the GUI checkpoint dialog boxes.

Note that you can associate the same property check with more than one class by repeating the **gui_ver_add_check_to_class** function for each GUI object class.

Example

In the following example, the Absolute_x check is added for the push_button class:

gui_ver_add_check_to_class ("push_button", "Absolute_x");

The *push_button* parameter defines the GUI object class, while *Absolute_x* defines the custom property check associated with it.

The following Check GUI dialog box displays the standard checks for push_button class objects:
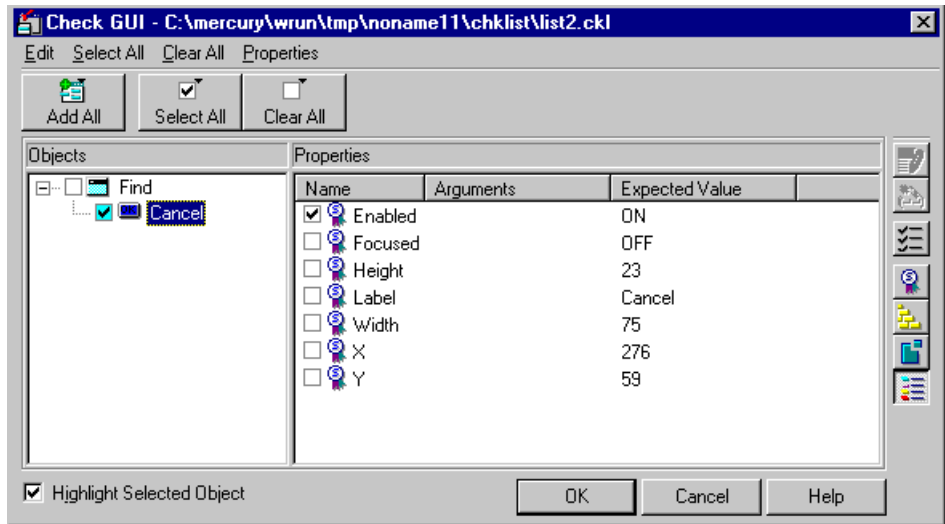
The following Check GUI dialog box displays a custom check as well as the standard checks for push_button class objects:

Custom check
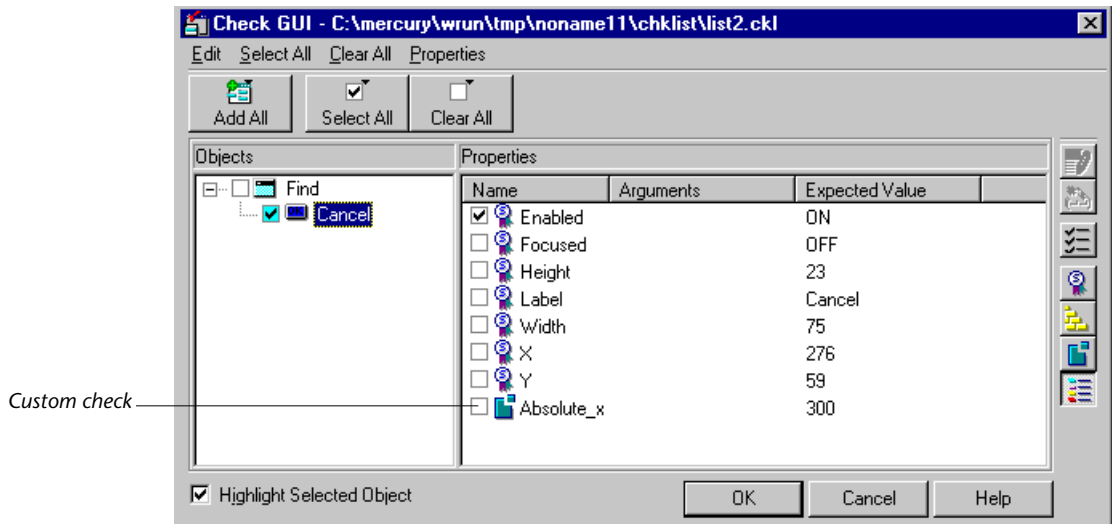
For additional information about the **gui_ver_add_check** and the **gui_ver_add_check_to_class** functions, refer to the *TSL Reference*.

Modifying the Default Checks for a GUI Object Class

You can modify the default property checks for a GUI object class. Similarly, you can define your custom property checks as default checks for a GUI object class. For example, by default, WinRunner checks only whether a push button is enabled. You can modify the default checks for the push_button class to include your custom check, Absolute_x.

To define a custom property check as a default property check, you use the TSL function **gui_ver_set_default_checks.** This function overwrites all previous default property checks; when using it, you must include all the property checks that you want to set as defaults for the GUI object class. Note that you can define additional standard property checks as default property checks using the same function.

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

gui_ver_set_default_checks (*class*, *check_name*$_1$...*check_name*$_n$ **);**

➤ *class* is the name of either the MSW_class or the standard class for which you want to set the default checks.

➤ *check_name*$_{1-n}$ are the names of the property checks to be set as defaults.

Example

In the following example, the Enabled and Absolute_x checks are set as the default checks for the push_button class.

gui_ver_set_default_checks ("push_button", "Enabled Absolute_x");

In the following dialog box, the custom Absolute_x property check is added.
The standard default property check for the push_button class, Enabled, is
selected:

Standard default check
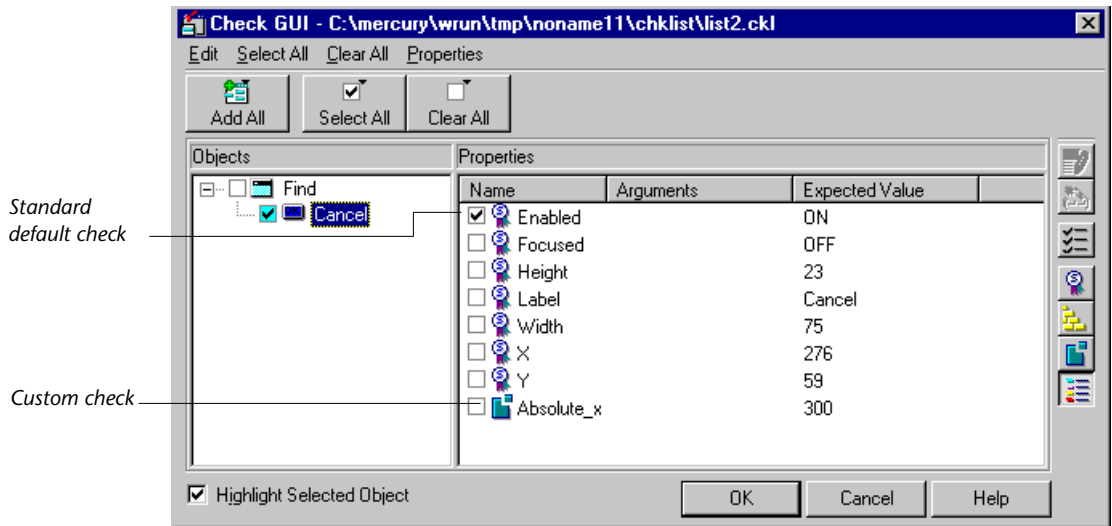
Custom check

In the following dialog box, both the custom Absolute_x property check and
the standard Enabled check are default property checks, i.e., they are
selected by default:

Custom check
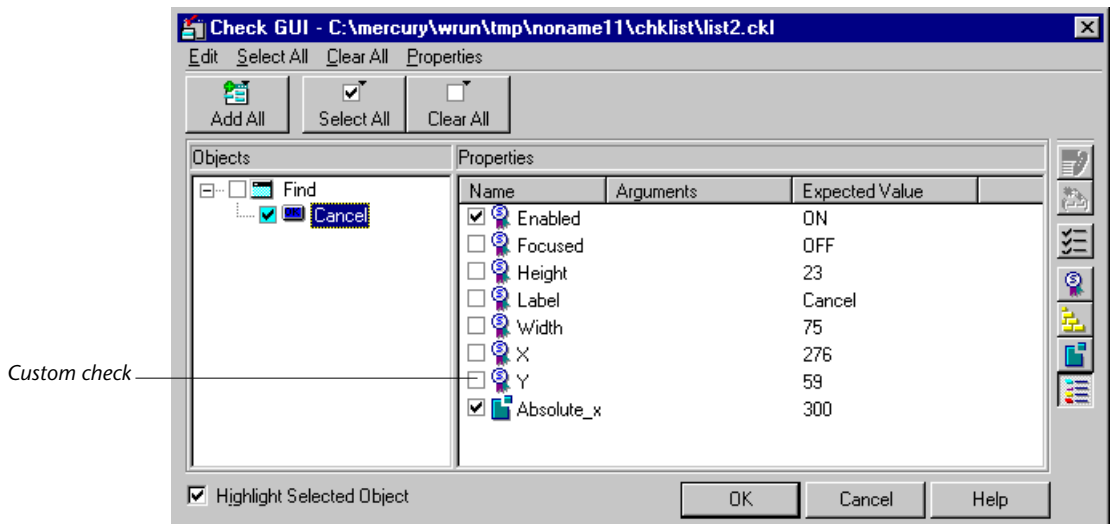
Note that when you define more than one default property check, separate the default property checks with spaces, as in the example above.

For additional information about the **gui_ver_set_default_checks** function, refer to the *TSL Online Reference*.

3

Creating GUI Checks for Custom Objects

You can create custom GUI object classes for verification, and develop checks for each custom class.

This chapter describes:

➤ Adding a Custom GUI Object Class for Verification

➤ Defining a Custom Check for a Custom GUI Object Class

About Creating GUI Checks for Custom Objects

Many applications contain GUI objects that do not belong to any of WinRunner's standard GUI object classes. By default, WinRunner recognizes these objects as belonging to the generic *object* class. If your application contains such objects, you can enhance your capability to check these objects by creating custom verification classes for them. You then develop property checks and GUI checkpoint dialog boxes for the new custom classes.

You use the standard WinRunner GUI checkpoint dialog boxes to check the custom objects. You can add custom property checks to the standard dialog boxes, as required. For example, you may have objects in your application that are classified by WinRunner as belonging to the generic "object" class. You can create a custom class called "pbTool" for these objects. The pbTool class will then have its own set of property checks which will be displayed in the standard GUI checkpoint dialog boxes.

Initially, the property checks displayed in these dialog boxes are the same as those displayed as for objects of the generic "object" class. Once you associate custom property checks with a custom class, these property checks are displayed whenever you create or edit a GUI checkpoint on objects belonging to this custom class.

When WinRunner does not recognize the class of a GUI object, in this case an object of the "pbTool" class, it assigns it to the generic "object" class. The GUI checkpoint dialog boxes display the property checks associated with the generic object class. For a list of property checks associated with the generic object class, refer to the "Checking GUI Objects" chapter in the *WinRunner User's Guide*.

You can add custom checks for this custom class, as described in this chapter. Develop your own GUI checkpoint dialog boxes with a customized user interface. For more information, see Chapter 4, "Creating GUI Checks: Advanced Topics."

To add a new GUI object class for verification, and then develop and specify the checks for it, you perform the following steps:

1 Add a new custom GUI object class for verification.

2 Create a capture function to establish the expected and actual results of the check.

3 Create a comparison function to compare the expected and the actual results.

4 Register the check that is defined by the capture and comparison functions.

5 Associate the new check with the new custom class.

6 Set the default checks for the new class.

The capture and comparison functions that you develop must be compiled before they can be used. Although you can do this by running the functions from a test script, it is recommended that you include all of them in a compiled module and load the module from a startup test. This makes the functions available in all your WinRunner sessions. For more information on compiled modules, refer to the "Creating Compiled Modules" chapter in the *WinRunner User's Guide*.

You can use WinRunner's Function Generator to generate all the required function calls, and then insert them directly into your test scripts. You can find the functions in the "GUI verification" category of the Function Generator. For more information on automatically generating and inserting functions, refer to the "Generating Functions" chapter in the *WinRunner User's Guide*.

Adding a Custom GUI Object Class for Verification

If you are implementing property checks for an object that does not belong to a standard WinRunner GUI object class, you must first define a new verification class for the object and then specify the property checks for the object class. You define a new class using the **gui_ver_add_class** function. This function has the following syntax:

gui_ver_add_class (*class_name* **[,***ui_function***] [,***default_check_function***]);**

➤ *class_name* is either the MSW_class property or the standard class property of the object. Use the GUI Spy to find the MSW_class property. For information on using the GUI Spy, refer to the "Configuring the GUI Map" chapter in the *WinRunner User's Guide*.

➤ The optional *ui_function* parameter enables you to develop and display the GUI checkpoint dialog boxes with a customized user interface. You create your own check dialog box only if this will enable you to more easily select the checks for a given checkpoint. For more information, see Chapter 4, "Creating GUI Checks: Advanced Topics."

You can use the **Property List** button in the GUI checkpoint dialog boxes to call the *ui_function* parameter. Note that this button is displayed only if at least one object in the Objects pane of the dialog box belongs to a class for which the *ui_function* parameter has been defined using the **gui_ver_add_class** function.

If you do not specify a *ui_function*, the set of property checks displayed for the new class will have the same checks as those displayed for the generic "object" class. For information on associating property checks with a class, see "Associating a New Property Check with a GUI Object Class" on page 20.

➤ The optional *default_check_function* parameter enables you to specify the runtime default checks for the new class. You use the *default_check_function* parameter only if you specify a *ui_function*, and you want to override the default checks. For more information, see Chapter 4, "Creating GUI Checks: Advanced Topics."

Note that by default, the property checks displayed in the GUI checkpoint dialog boxes for a custom class are the same as those displayed for the generic object class. You can add your own custom checks for the custom class, so that they will be displayed for that class in the GUI checkpoint dialog boxes.

The Check GUI dialog box below displays the default checks for any custom object, which is associated with the generic "object" class:
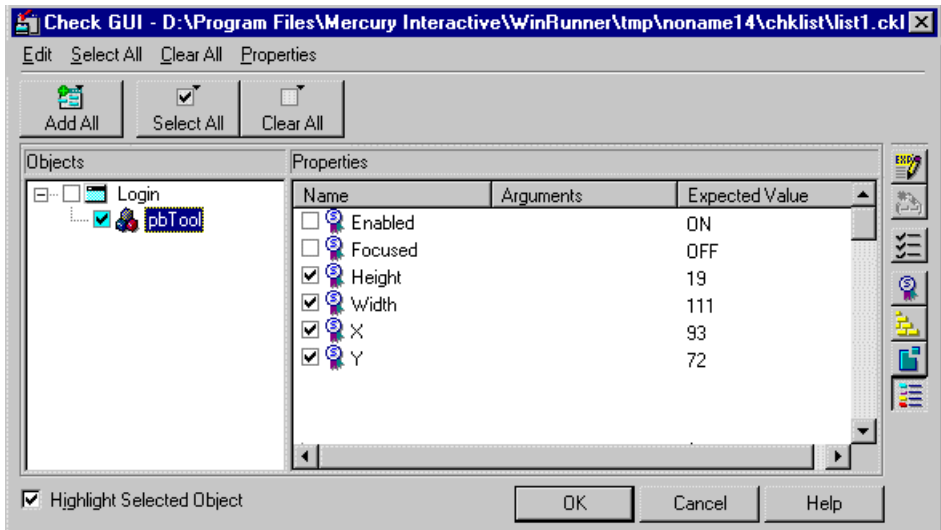
You can add a custom check for this custom class, as shown in the dialog box below:

Custom check
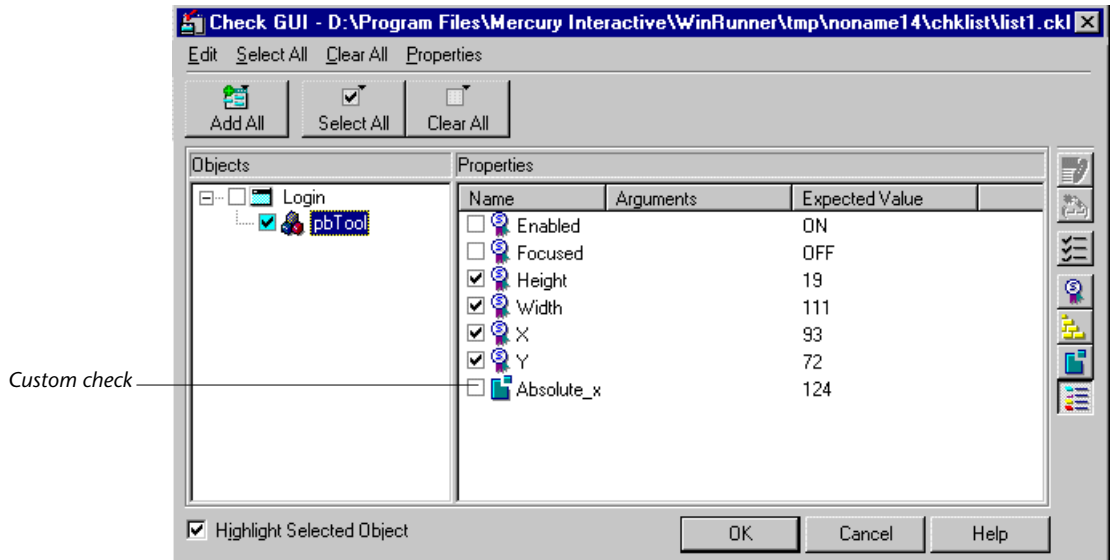
You can also map custom objects to standard object classes. For additional information, refer to the "Configuring the GUI Map" chapter in the *WinRunner User's Guide*.

Example: Adding a New GUI Object Class for Verification

You can use WinRunner's GUI Spy to see the properties of the toolbar in the Paint application. For information on using the GUI Spy, refer to the "Configuring the GUI Map" chapter in the *WinRunner User's Guide*. WinRunner recognizes the toolbar as belonging to the generic "object" class. The MSW_class property of the toolbar is "pbTool." You can create a custom class for the toolbar using the following statement:

gui_ver_add_class ("pbTool");
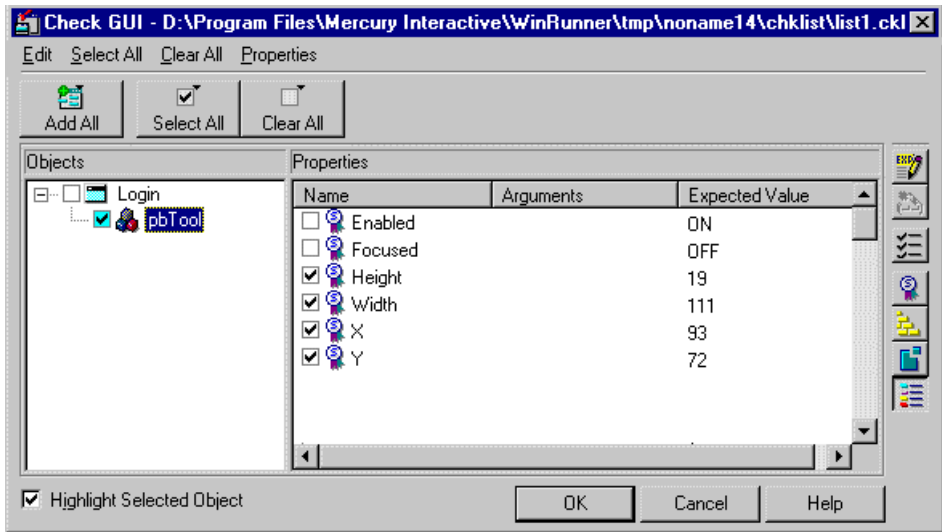
By default, the property checks displayed in the GUI checkpoint dialog boxes for the "pbTool" class are the same as those displayed for the generic "object" class.

Defining a Custom Check for a Custom GUI Object Class

Once you have created a custom class for verification, you can add property checks to the custom class.

To add the property checks to a custom class, perform the tasks listed below. For details of each of these tasks, see Chapter 2, "Creating Custom GUI Checks for Standard Objects."

1 Create a capture function.

The capture function establishes and stores the expected and actual results for the property check.

2 Create a comparison function.

After the capture function has determined the expected and actual results for a property check, WinRunner verifies the results to determine whether the check passed. To verify a check, you can either create your own comparison function, or use WinRunner's standard comparison function, **default_compare_func**.

3 Register the new check.

Once you have created a new GUI object class for verification and developed and compiled the capture and comparison functions, you must register the new check that the functions define. This is done using the **gui_ver_add_check** TSL function.

4 Associate the new property check with a class.

Having registered the new property check, you associate it with a custom GUI object class for verification. By associating the new property check with a class, you add the property check to the list of property checks displayed for that class in the GUI checkpoint dialog boxes. You associate the property check with a class using the **gui_ver_add_check_to_class** TSL function.

5 Set the default checks.

You set the default checks for a custom GUI object class using the **gui_ver_set_default_checks** TSL function.

For additional information about the TSL functions described above, refer to the *TSL Online Reference*.

4

Creating GUI Checks: Advanced Topics

You can create your own user interface for the GUI checkpoint dialog boxes for objects of a custom GUI object class. In addition, you can implement your own results display utility for the check.

This chapter describes:

➤ Adding a New GUI Object Class for Verification

➤ Creating Capture and Comparison Functions

➤ Registering the New Check

➤ Setting the Default Checks

➤ Implementing Advanced GUI Checking

About Advanced Topics in Creating GUI Checks

If your application contains objects that do not belong to a standard WinRunner GUI object class, you can create a custom class in order to check them. For each custom class, you can either:

➤ Use the standard WinRunner GUI checkpoint dialog boxes, which include the standard checks for the generic "object" class, as well as any custom checks you add. For more information, see Chapter 3, "Creating GUI Checks for Custom Objects."

➤ Develop your own GUI checkpoint dialog boxes with a customized user interface and an associated results display facility, as described in this chapter.

To add a new GUI object class, and then develop a custom user interface and custom results display utility, you perform the following steps:

1 Define a new custom GUI object class, and develop GUI checkpoint dialog boxes with a customized user interface.

2 Create a capture function to establish the expected and actual results of the check.

3 Create a comparison function to compare the expected and the actual results.

4 Register the property check that is defined by the capture and comparison functions.

5 Set the default checks for the new class.

Adding a New GUI Object Class for Verification

You define a new GUI object class for verification using the function **gui_ver_add_class**. For details on defining a new class for verification, see Chapter 3, "Creating GUI Checks for Custom Objects."

Note that when you use the **gui_ver_add_class** function, the *ui_function* parameter is the name of the user-defined function that enables you to develop and display the GUI checkpoint dialog boxes with a customized user interface. The Check GUI dialog box is displayed when you double-click a GUI object when creating a GUI checkpoint. You create GUI checkpoint dialog boxes with a customized user interface to enable you to more effectively select the checks for a given checkpoint. For example, suppose you have a custom class for *tables*. You could develop GUI checkpoint dialog boxes with a customized user interface that enable you to easily select which columns of the table to compare, and on what basis to compare the columns.

The *ui_function* parameter has the following syntax:

function **ui_function (** *in window, in object, inout check_list, inout arg_list* **);**

➤ *window* is the description of the window in which the object exists.

➤ *object* is the description of the object selected by the user.

➤ *check_list* serves two functions: The default checklist for the object is passed to the function when the function is called. The function must pass a new checklist as output. The checklist consists of one or more check names, separated by spaces or commas.

➤ *arg_list* is a string that you want to return as an output. It is passed as a parameter to each property check's capture and comparison functions. You can store the *arg_list* parameter in a file using the file name supplied by the *arg_list* parameter.

You can use the **Property List** button in the GUI checkpoint dialog boxes to call the *ui_function* parameter. Note that this button is displayed only if at least one object in the Objects pane of the dialog box belongs to a class for which the *ui_function* parameter has been defined using the **gui_ver_add_class** function.

The *default_check_function* parameter has the following syntax:

function **default_check_function (** *in window, in object, inout check_list, inout arg_list* **);**

The parameters of *default_check_function* are the same as those for *ui_function*, described above.

For an example of how to use the **gui_ver_add_class** function to develop customized GUI checkpoint dialog boxes, see "Implementing Advanced GUI Checking" on page 37.

Creating Capture and Comparison Functions

You create a capture function to establish and store the expected and actual results for the custom property check. After the capture function has determined the expected and actual results for a check, WinRunner verifies the results to determine whether or not the check passed. To verify a check, you can either create your own comparison function, or use WinRunner's standard comparison function, **default_compare_func**.

For details on capture and comparison functions, see Chapter 2, "Creating Custom GUI Checks for Standard Objects."

For an example illustrating the use of capture and comparison functions, see "Implementing Advanced GUI Checking" on page 37.

Registering the New Check

Once you have created the capture and comparison functions, you register the new property check that the functions define. For details on registering the check, see Chapter 2, "Creating Custom GUI Checks for Standard Objects."

Note that when you use the **gui_ver_add_check** function, the *display_function* parameter enables WinRunner to use a custom display utility to view the results of the check. For example, if you select a property check in the GUI Checkpoint Results dialog box that has an associated display function, the Display button is enabled, as shown below. You click the Display button to display the results of the check using your display function. If you do not enter a *display_function*, then WinRunner uses its default result display facilities.
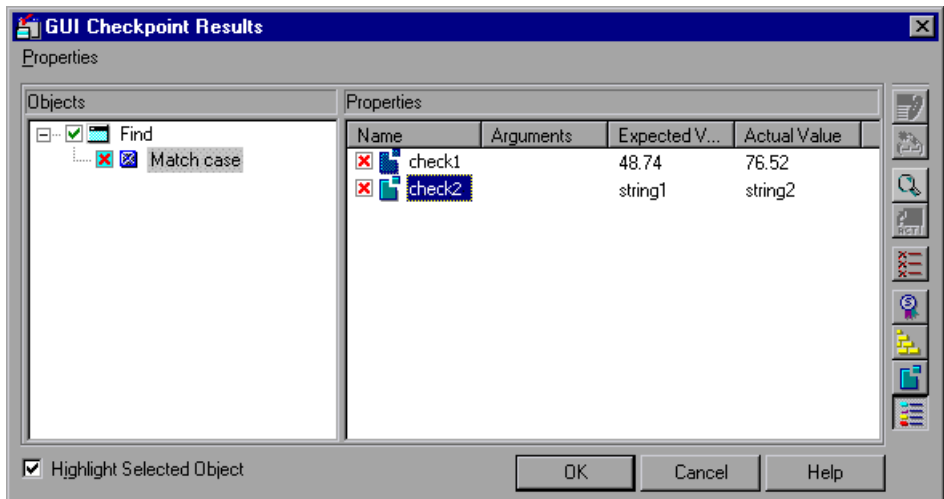
Note that since "check2" has an associated display function, the Display button is enabled.

The *display_function* parameter has the following syntax:

function **display_function (** *in expected, in actual, in result, in diff* **);**

➤ *expected* is the expected result string or filename.

➤ *actual* is the actual result string or filename.

➤ *result* is the result of the compare function: 0 for a successful comparison, any other value for a mismatch.

➤ *diff* is a string, received from the comparison function and may contain a filename or any other information needed by the display function.

Setting the Default Checks

To set the default checks for the new check, you use the **gui_ver_set_default_checks** function. For details on setting the default checks, see Chapter 2, "Creating Custom GUI Checks for Standard Objects."

Implementing Advanced GUI Checking

The following example illustrates the use of most GUI verification customization features. For the sake of simplicity, all the checks simply return random numbers. The example adds a custom class, *AfxWnd*, and then adds the checks for the class. The *ui_function* parameter uses WinRunner's **pause_test** function to create a simple output/input dialog box, as shown below.
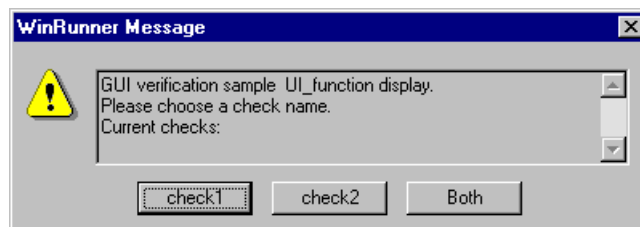
Dialog box with custom user interface developed using WinRunner's pause_test function

This example presents a mechanism that can be used for GUI verification customization. Using external DLLs, you can implement more sophisticated output/input screens based on this prototype.

This test is designed to operate on WinBurger, a sample application supplied with WinRunner. The *Reset* button in WinBurger belongs to the "AfxWnd" custom class, the class that is customized in this example. You can locate WinBurger in your *installation_directory\samples\bin\winbur* folder.

```
# load user-defined functions
reload("udf_gui");
```

```
# Add new verification class, "AfxWnd"
rc=gui_ver_add_class("AfxWnd", "reset_ui_func"); # adds class "AfxWnd"
```

```
# Register check1
rc=gui_ver_add_check("check1","check1_capt","compare1");
```

```
#Add check1 to class
gui_ver_add_check_to_class("AfxWnd","check1");
```

```
# Register check2
rc=gui_ver_add_check("check2","check2_capt","compare2","display_func2");
```

```
#Add check2 to class
gui_ver_add_check_to_class("AfxWnd","check2");
```

```
# Set default checks for new class
rc=gui_ver_set_default_checks("AfxWnd","check1");
-----------------------------------------------------------------------------------
```

```
# udf_gui
function reset_ui_func(window, object, inout checklist, out arglist)
{
    auto res=pause_test("GUI Verification Sample UI_function Display\nPlease
choose a check name.\nCurrent checks:" & checklist, "check1", "check2", "Both"
);
    if (res==0) {
        checklist = "check1";
        arglist = "User selected check1";
    }
    else
    if (res==1) {
```

```
        checklist = "check2";
        arglist = "User selected check2";
    }
    else
    if (res==2) {
        checklist = "check1 check2";
        arglist = "User selected check1 & check2";
    }
    else
        return -1;
    return 0;
}
```

Capture Function #1

```
function check1_capture(object, inout value)
{
    value = 100*rand();
    return 0;
}
```

Capture Function #2

```
function check2_capture(object, inout file)
{
    file_open(file,FO_MODE_WRITE);
    file_printf(file, "%"S",""The result of check2 was sent to a file.\n\nThe result of
check2 is: " & 100*rand() )");
    file_close(file);
    return 0;
}
```

Comparison Function #1

```
function compare1(exp_val, act_val, arglist, inout diff_file)
{
    diff_file = "";
    if (exp_val != act_val) {
        return E_DIFF;
    }
    return E_OK;
}
```

Comparison Function #2

```
function compare2(exp_file, act_file, arglist, inout diff_file)
{
    auto exp_buf, act_buf;
    read_file(exp_file, exp_buf);
    read_file(act_file, act_buf);
    if (exp_buf != act_buf || arglist != "") {
        file_open(diff_file,FO_MODE_WRITE);
        file_printf(diff_file,"Difference forced !");
        file_close(diff_file);
        return 1;
    }
    diff_file = "";
    return E_DIFF;
}
```

Display Function

```
function display_func2(exp_file, act_file, result, diff_file)
{
    auto exp_buf, act_buf, diff_buf;
    read_file(exp_file, exp_buf);
    read_file(act_file, act_buf);
    read_file(diff_file, diff_buf);
    pause_test("\nExpected: " & exp_buf & "\nActual: " & act_buf & "\n\nResult: "
& result & "\nDiff: " & diff_buf, "OK", "Cancel", "Close");
    return 0;
}
```

```
function read_file(name, out buf)
{
    auto    tmp;
    buf = "";
    file_open(name,FO_MODE_READ);
    if (name != "") {
        while (file_getline(name,tmp)) {
            buf = buf & tmp;
        }
        file_close(name);
    }
}
```

Part II

Customizing Recording

5

Customizing Recorded Statements

When you record operations on custom GUI objects, the resulting test script contains generic **obj_** TSL statements. You can make the test script easier to read by creating custom record functions.

This chapter describes:

➤ Understanding Custom Record Functions

➤ Developing a Custom Record Function

➤ Associating a Custom Record Function with a GUI Object Class

➤ Developing a Custom Execution Function

➤ Example of a Custom Record Function

About Customizing Recorded Statements

Many applications contain custom GUI objects—objects that do not belong to any of WinRunner's standard GUI object classes. Because WinRunner is not familiar with the behavior of such custom objects, whenever you operate one of them, WinRunner records a generic **obj_mouse** statement into your test script. Because these **obj_mouse** statements are generic, two problems can arise:

➤ The recorded statements are not descriptive, and the test script is therefore difficult to read and analyze.

➤ The recorded statements do not fully describe the operations that were performed. When you run the test, WinRunner does not correctly duplicate the recorded operations.

By implementing custom record functions, you can resolve both these problems.

Custom record functions enable you to specify the statements that WinRunner records in place of generic **obj_** statements. That is, you specify the statement to be recorded when you perform a specific operation on a GUI object belonging to a custom object class.

To implement a custom record function, you perform the following tasks:

1 Develop a custom record function.

2 Associate the custom record function with a custom GUI object class.

3 Implement a custom execution function, if required.

Note: You can combine a custom record function with a logical name function for a given GUI object class, thereby further improving the readability of the statements that WinRunner records into your test scripts. Logical name functions enable WinRunner to assign descriptive logical names to custom GUI objects. For more information, see Chapter 7, "Customizing Assigned Logical Names."

Understanding Custom Record Functions

To illustrate the use of a custom record function, consider the following scenario:

Note: The scenario assumes that you have not installed WinRunner with support for Visual Basic. If you have the Visual Basic support installed, then custom record function described in the scenario will already be implemented for the spinbutton class.

Assume that your application contains a custom "spinbutton." If you click on the upper arrow of the spinbutton, the associated counter is incremented by one. If you click on the lower arrow, the counter is decremented by one. If you click on either arrow and hold down the mouse button, the counter is incremented or decremented continuously, as long as the button is held down.

Increment arrow

Decrement arrow

Counter
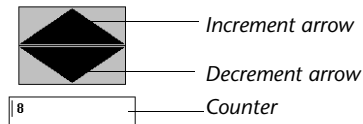
Because this spinbutton does not belong to a standard GUI object class, you create a custom class for the spinbutton, called simply "spinbutton." Suppose you record three different mouse clicks on the spinbutton. WinRunner records statements similar to the following into your test script:

Operation Performed	Recorded Test Script Statement	Counter Before	Counter After
click on upper arrow	obj_mouse_click ("SpinButton", 50, 22, LEFT);	1	2
click on lower arrow	obj_mouse_click ("SpinButton", 50, 58, LEFT);	2	1
click on upper arrow, mouse button held down	obj_mouse_click ("SpinButton", 50, 22, LEFT);	1	6

Improving the Readability of Recorded Statements

The above recorded statements are identical, except for the coordinates of the mouse click. It is therefore difficult to distinguish what operation is recorded with each statement. That is, was the increment arrow pressed, or the decrement arrow? Was there a simple click, or was the mouse button held down for a period of time?

You can make the recorded test script easier to read and analyze by implementing a custom record function—thereby enabling WinRunner to record more descriptive statements.

Recorded Test Script Statement	Operation Performed
spin_up ("SpinButton", 50, 22, LEFT);	click on upper arrow
spin_down ("SpinButton", 50, 58, LEFT);	click on lower arrow

From the above "customized" statements, it is immediately apparent that the spin_up statement indicates a mouse click on the upper (increment) arrow, and that the spin_down statement indicates a mouse click on the lower (decrement) arrow.

Improving Execution Accuracy

WinRunner executes the first two "simple" mouse clicks as required. However, the third statement, which represents the mouse button being pressed and held down, is not executed correctly. The third mouse click is recorded with a generic **obj_mouse_click** statement which contains no information describing how many times the arrow was activated while the mouse button was held down. WinRunner therefore executes a simple mouse click, and the counter increments from 1 to 2, and not from 1 to 6, as is required.

By implementing a custom record function you can resolve the execution difficulty. Custom record functions ensure that the statements that WinRunner records fully describe the operations that you perform. This enables WinRunner to execute the recorded statements as required.

The table below shows modified spin_up and spin_down statements. The modification includes a parameter that defines how many times each arrow was activated while the mouse button was held down.

Operation Performed	Recorded Test Script Statement	Counter Before	Counter After
click on upper arrow	spin_up ("SpinButton", 1);	1	2
click on lower arrow	spin_down ("SpinButton", 1);	2	1
click on upper arrow, mouse button held down	spin_up ("SpinButton", 5);	1	6

Note that custom record functions generate functions that are not standard to WinRunner. A *custom execution function* is required to execute a non-standard function. For more information, see "Developing a Custom Execution Function" on page 54.

Combining a Custom Record Function with a Logical Name Function

You can further improve the readability of your test scripts by implementing a custom record function together with a logical name function. Logical name functions enable WinRunner to assign descriptive logical names to custom GUI objects. For more information, see Chapter 7, "Customizing Assigned Logical Names."

Developing a Custom Record Function

In order to customize recorded statements in a test script, you create a custom record function that returns a string when you perform a specific operation on an object in a custom object class. WinRunner uses this string as the basis of the statement it records into the test script.

For example, suppose you implement a custom record function that returns a string when you click on a custom spinbutton. Before implementing the function, WinRunner records a statement similar to the following:

obj_mouse_click("spinbutton",50,100);

After implementing the record function, WinRunner records a statement similar to the following:

spin_up("spinbutton", 1);

Writing the Custom Record Function

You write the custom record function in C-language, and compile and link it into a DLL. You supply the name of the DLL when you associate the custom record function with a GUI object class. See "Associating a Custom Record Function with a GUI Object Class" on page 53.

The custom record function has the following prototype:

int **function** (HWND FAR* *phWnd*, UINT *msg*, WPARAM *wParam*, LPARAM *lParam*, char* *str*, int *size*);

Parameters

phWnd	The pointer to the handle of the GUI object that is being acted upon. You can modify the contents of the pointer to record an operation on a different GUI object.
msg	The Windows message received.
wParam	The first message parameter.
lParam	The second message parameter.

str	A buffer allocated by WinRunner. The record function assigns the string to be recorded into the test script to the allocated buffer.
size	The size of the *str* buffer, in bytes.

The record function assigns the string that WinRunner will record into the test script, to the *str* parameter. You use WinRunner's "%m" format to create the string, by substituting "%%m" for the GUI object's logical name. For example, the custom record function could assign the following string to the *str* parameter:

spin_up ("%%m", 1)

Do not include a semicolon (;) at the end of the string—WinRunner adds a semicolon when it records the statement into the test script. WinRunner replaces "%m" with the logical name of the GUI object when the statement is recorded into the test script.

Returning a Value

The record function returns a value that describes the *sending mode* of the function. For example, the following statement could be included in a custom record function:

return(SEND_LINE);

The various sending mode options that you can return are described below:

➤ SEND_LINE

Instructs WinRunner to record the string returned in the *str* parameter into the test script. If a string has been previously stored, WinRunner records that string first into the test script.

➤ KEEP_LINE

Instructs WinRunner to store the string returned in the *str* parameter, and start the timer. If a string has been previously stored, WinRunner records it into the test script. If time-out occurs and no further mouse input is detected, WinRunner records the new string into the test script. Use KEEP_LINE after detecting a single mouse click. WinRunner waits to establish if the single mouse will be followed by another click during the time-out period, thereby producing a double-click.

➤ KEEP_LINE_NO_TIMEOUT

Instructs WinRunner to store the string returned in the *str* parameter. If WinRunner has previously stored a string, then WinRunner records that string into the test script.

➤ REPLACE_AND_SEND_LINE

Instructs WinRunner to record the string returned in the *str* parameter into the test script. If a string has been previously stored, WinRunner deletes that string.

➤ REPLACE_AND_KEEP_LINE

Instructs WinRunner to store the string returned in the *str* parameter, and start the timer. If a string has been previously stored, WinRunner deletes it. If time-out occurs and there is no further mouse input, WinRunner records the string into the test script.

➤ CLEAN_UP

Instructs WinRunner to record the string stored in the *str* parameter into the test script, if a stored string exists.

➤ NO_PROCESS

If the *str* parameter is empty, NO_PROCESS instructs WinRunner to record the default function for the GUI object.

If the *str* parameter is not empty, NO_PROCESS instructs WinRunner not to record any function.

Adding Windows Messages

Although WinRunner monitors all Windows messages, only a small number of the many Windows messages are actually processed. That is, WinRunner ignores all but a few messages. The record functions that you implement may require that WinRunner process additional messages. You use the **add_record_message** TSL function to specify which additional messages to include. The **add_record_message** function has the following syntax:

add_record_message (*message_number* **);**

➤ *message_number* is the number or identifier of the additional Windows message that you want WinRunner to process.

For example, the following statement instructs WinRunner to add the WM_MOUSEMOVE message to the list of messages that it processes.

add_record_message(512);

For an example of a custom record function, see "Example of a Custom Record Function" on page 55.

Associating a Custom Record Function with a GUI Object Class

You use the **add_cust_record_class** function to associate the custom record functions with a GUI object class. The **add_cust_record_class** function has the following syntax:

add_cust_record_class (*MSW_class*, *dll_name* **[,** *rec_func* **]**
   [, *log_name_func* **]);**

➤ *MSW_class* is the MSW_class of the custom objects with which the custom record function is associated.

➤ *dll_name* is the full path and filename of the DLL in which you compiled and linked the custom record function. If a logical name function also exists for the GUI object class, that function is also contained in this DLL. For more information, see Chapter 7, "Customizing Assigned Logical Names."

> ➤ *rec_func* is the name of the record function in the DLL. The record function returns the string that WinRunner records into the test script.

> ➤ *log_name_func* is the name of the logical name function (if one exists) that is included in the DLL. The *log_name_func* function supplies a custom logical name for a custom GUI object in class *MSW_class*. For more information, see Chapter 7, "Customizing Assigned Logical Names."

In the following example, the **add_cust_record_class** function adds a custom record function, *SpinHighLevelRec*, for the SpinButton class.

```
add_cust_record_class("SpinButton", "c:\\arch\\vb_util.dll",
    "SpinHighLevelRec", " ");
```

Developing a Custom Execution Function

If you implement a custom record function that generates a custom statement, you must develop a custom execution function to enable WinRunner to execute the recorded statement. For example, assume you develop a custom record function that records the following function into your test script:

```
custom_function(2,2);
```

Because custom_function is a call to a user-defined function, WinRunner does not recognize it, and consequently cannot execute the function. You develop a user-defined function that defines what WinRunner must do each time it executes a custom_function statement.

Custom execution functions obtain information from your application about the custom object, such as its state or position, and then move the mouse cursor to the required location and enter mouse or keyboard input. A number of TSL functions, such as **obj_get_info**, can be used to retrieve information about the object. Other TSL functions such as **click**, **obj_mouse_drag**, and **move_locator_abs** can be used to execute the function.

The following example shows the TSL implementation of the **spin_up** execution function. You may want to base the implementation of your execution functions on the example below.

```
public function spin_up(win, times){
    auto hWnd;
    auto res;

    # get GUI object handle to send to DLL
    res = obj_get_info(win, "handle", hWnd);
    if(res != E_OK)
        return(res);

    # call DLL, _spin_up
    res = _spin_up(hWnd, times);

    # internal TSL function, called if _spin_up fails
    if(res != E_OK)
        process_return_value(res);
    return(res);
}
```

Example of a Custom Record Function

The following example illustrates the implementation of a custom record function for the Visual Basic control, ThunderlistBox. You may want to base the implementation of your custom record functions on this example. The file cust_rec.h is included to set the return values for the record function.

filename: cust_rec.h

```
// Return values for recording function
#define SEND_LINE       0
#define KEEP_LINE       1
#define REPLACE_AND_SEND_LINE   2
#define REPLACE_AND_KEEP_LINE   3
#define CLEAN_UP        4
#define KEEP_LINE_NO_TIMEOUT    5
#define NO_PROCESS  6
```

filename: cust_rec.c

```
#define EXPORTED _far _pascal __export
#include <windows.h>
#include "cust_rec.h"
#include <windowsx.h>   // Windows Messages Cracker

#define MAXKEYS     9
BOOL isMouseUponObject(HWND hwin, LPARAM lParam);
WORD GetListKey       (HWND hwin, WPARAM wParam);

// Custom Recording for ThunderListBox
//--------------------------------------------------------------------------
//                    ThunderListBox
//
// Implementation of the Visual Basic ListBox
// All the ListBox commands are written by the Windows Messages Cracker's
// format
//
// NOTE:
// WM_KEY* messages are received not as in the Windows documentation,
// rather:
//    wParam = scan code; and for extended keys, top most bit is set.
//    lParam does not contain interesting information.
//--------------------------------------------------------------------------

int EXPORTED ThunderListBox(HWND FAR* pwin, UINT msg,
                WPARAM wParam, LPARAM lParam,
                LPSTR rec_str, int len)
{
   static bMouseDown   = FALSE;
   static bPushKeyDown = FALSE;
   WORD    wVirtKey;
   int     nReturn = SEND_LINE;
   int     nItemSel;
   char    szBuff[255];

   UINT ss;

   HWND win = *pwin;
```

```
switch (msg) {
case WM_LBUTTONDBLCLK:
  nItemSel = ListBox_GetCurSel(win);
  if (nItemSel != LB_ERR) {
    // if something is being selected - get the selected string
    ListBox_GetText(win, nItemSel, szBuff);
    // and send the command line
    wsprintf(rec_str,
      "ActivateThunderListItem (\"%%m\", \"%s\")", (LPSTR)szBuff);
    // replacing the previous one which was "SelectThunderListItem"
    nReturn = REPLACE_AND_SEND_LINE;
  }
  break;

// set the bMouseDown Flag
case WM_LBUTTONDOWN:
  if (isMouseUponObject(win, lParam))
    bMouseDown = TRUE;
  break;

case WM_LBUTTONUP:
  if (bMouseDown) {   // if the bMouseDown flag has been set
    bMouseDown = FALSE;
    nItemSel = ListBox_GetCurSel(win);
    if (nItemSel != LB_ERR) {
      // and something is being selected - get the selected string
      ListBox_GetText(win, nItemSel, szBuff);
      // and send the command line
      wsprintf(rec_str,
        "SelectThunderListItem (\"%%m\", \"%s\")",
        (LPSTR)szBuff);
      // and keep it for the DBLCLICK case
      nReturn = KEEP_LINE;
    }
  }
  break;

case WM_KEYDOWN:
  // get the real VirtKey value
```

```
            // and if it's valid - set the bPushKeyDown flag
            ss = HIWORD(lParam);
            ss = LOBYTE(ss);
            ss = MapVirtualKey(ss, 1);

            if (GetListKey (win, wParam) != 0xFFFF)
                bPushKeyDown = TRUE;
            break;

        case WM_KEYUP:
            // get the real VirtKey value
            wVirtKey = GetListKey (win, wParam);
            if (bPushKeyDown && (wVirtKey != 0xFFFF)) {
                // if it's valid and the bPushKeyDown flag has been set -
                // reset the flag
                bPushKeyDown = FALSE;

                nItemSel = ListBox_GetCurSel(win);
                if (nItemSel != LB_ERR) {
                    // if something is being selected - get the selected string
                    ListBox_GetText(win, nItemSel, szBuff);
                    // and send the appropriate command line
                    // - Activate or Select
                    if (wVirtKey == VK_RETURN)
                        wsprintf(rec_str,
                            "ActivateThunderListItem (\"%%m\", \"%s\")",
                            (LPSTR)szBuff);
                    else
                        wsprintf(rec_str,
                            "SelectThunderListItem (\"%%m\", \"%s\")",
                            (LPSTR)szBuff);
                }
            }
            break;

        default:
            break;
    }
```

```
    return (nReturn);
}


//----------------------------------------------------------------------
// isMouseUponObject
// Check if the Mouse is placed upon the hwin Window Rectangle
//----------------------------------------------------------------------

BOOL isMouseUponObject(HWND hwin, LPARAM lParam)
{
    RECT    rect;
    POINT   ptMouse;
    BOOL    bResult;

    // Get the local mouse coordinates
    ptMouse.x = LOWORD(lParam);
    ptMouse.y = HIWORD(lParam);

    // Get the Object's Window coordinates
    GetWindowRect(hwin, (RECT FAR*)&rect);

    // Convert the local mouse coordinates to the Window one
    ClientToScreen(hwin, (POINT FAR*)&ptMouse);

    // test if the mouse upon the Object Window's rectangle
    bResult = PtInRect((const RECT FAR*)&rect, ptMouse);

    return bResult;
}


//----------------------------------------------------------------------
// GetListKey
// Recognize if the Key message corresponds to any of the Virtual Keys:
// VK_RETURN, VK_UP, VK_RIGHT, VK_LEFT, VK_DOWN, VK_HOME,
// VK_END, VK_PRIOR, VK_NEXT,
//
// Assume that wParam and lParam have been changed and for some reason
// for the Keyboard's keys (except the ENTER) and for KeyPad's ENTER there
// is a topmost bit being added to the wParam value
```

```
//------------------------------------------------------------------------

WORD GetListKey(HWND hwin, WPARAM wParam)
{
   int    i;
   BOOL   bKeyPad,
          bNumLock;

   // regular movement & executive keys
   WORD wVirtKeys[MAXKEYS] =
     {VK_RETURN, VK_UP, VK_RIGHT, VK_LEFT,
      VK_DOWN, VK_HOME, VK_END, VK_PRIOR, VK_NEXT};

   // KeyPad's key values have been recognized by debugging
   bKeyPad = ((wParam >= 0x47) && (wParam <= 0x51));

   // check the NUMLOCK key's toggle - test the lowest bit
   bNumLock = 0x01 & GetKeyState(VK_NUMLOCK);

   if (!bKeyPad || (bKeyPad && !bNumLock)) {
      // process all the keys but the <NUMLOCK<->KEYPAD> case
      // (except the KeyPad's ENTER)
      for (i = 0; i < MAXKEYS; i++) {
         // reset the wParam's topmost bit and test the Virtual Keys.
         if ((wParam & 0x7F) == MapVirtualKey(wVirtKeys[i], 0))
            // return the current Virtual Key's value
            return wVirtKeys[i];
      }
   }

   // return "nothing"
   return 0xFFFF;
}
```

6

Adding Custom Properties for GUI Objects

You can add your own properties to any GUI object class to improve WinRunner's ability to uniquely identify the GUI objects in your application.

This chapter describes:

➤ Developing a Query Function for a Custom Property

➤ Developing a Verification Function for a Custom Property

➤ Registering a Custom Property

➤ Assigning a Custom Property to a GUI Object Class

➤ Example of a Custom Property Function

About Adding Custom Properties for GUI Objects

WinRunner uniquely identifies each GUI object in your application by developing a physical description for the object. This physical description is made up of a list of the object's obligatory properties. When the obligatory properties do not provide unique identification, WinRunner includes optional properties into the physical description. If this still does not uniquely identify the object, then WinRunner includes a selector as well. See your *WinRunner User's Guide* for more information on how WinRunner identifies GUI objects.

The inclusion of optional properties and selectors can lead to long and complex physical descriptions. To improve the efficiency with which WinRunner identifies GUI objects, you can register your own user-defined or *custom* properties. For example, every GUI object created using Visual Basic is assigned the *vb_name* property. Within a given window, the *vb_name* property is always unique. If you have installed WinRunner with Visual Basic support, then the *vb_name* property is automatically added as a custom property. This enables WinRunner to more efficiently identify the GUI objects in your Visual Basic applications.

Like Visual Basic, most other development environments also assign properties to their GUI objects. By defining these properties as custom WinRunner properties, you enhance WinRunner's ability to uniquely identify GUI objects in your applications.

To add a custom property, you perform the following tasks:

1 Develop a query function to obtain the value of the custom property, whenever the value is required.

2 Develop or specify a function that verifies whether the custom property of a given GUI object has the required value.

3 Register the custom property.

4 Assign the custom property to a GUI object class.

Developing a Query Function for a Custom Property

You create a query function to evaluate and return the value of a custom property of a GUI object. For example, suppose you add the custom property *new_property*. It is the query function that actually evaluates and returns the value of *new_property* for a given GUI object, whenever the value is required.

You write the query function in C-language, and compile and link it into a DLL. You supply the name of the DLL when you register the custom property. See "Registering a Custom Property" on page 65.

The query function has the following prototype:

void **query_func** (HWND *hWnd*, LPSTR *str*, int *size*)

Parameters

hWnd	The handle of the GUI object for which the custom property is being evaluated.
str	A buffer allocated by WinRunner to which the query function assigns the value of the custom property.
size	The size of the *str* buffer in bytes.

The *query_func* function is a library-defined callback function that WinRunner calls whenever the value of a custom property is required. The *query_func* function is a placeholder for the library-defined function name. The actual name must be exported by including it in an EXPORTS statement in the library's module-definition (.DEF) file.

For an example of a query function for a custom property, see "Example of a Custom Property Function" on page 69.

Developing a Verification Function for a Custom Property

WinRunner uses a verification function to identify GUI objects in your application whose custom property has the required value.

Understanding the Verification Function

To understand the role of a verification function, consider the following scenario. Suppose that you open the GUI Map Editor, and select a custom object. The object is uniquely identified by the class property, object, and its custom property, *new_property*. Suppose that the value of *new_property* is *Object1*. The physical description of the object is therefore:

{class:"object", new_property:"Object1"}

When you click the Show button in the GUI Map Editor, WinRunner attempts to highlight the selected object. But first it must locate it.

To locate the required object, WinRunner systematically examines each object in your application, and enquires: Does this object belong to the "object" class? If not, WinRunner queries the next object. If that object belongs to the "object" class, WinRunner calls the query function to establish the value of the object's custom property, *new_property*. Having established the value of *new_property*, WinRunner calls the verification function to determine if this value matches the required value, Object1. If the verification function indicates a mismatch, WinRunner continues searching for the required object. When the query function returns a value of "Object1", the verification function indicates that the correct object is found, and WinRunner highlights the object.

Using Standard and Custom Verification Functions

The function that you use to verify the value of a custom property can be one of WinRunner's standard property verification functions, or your own custom verification function.

The standard property verification functions are:

➤ **string_verify**, which compares the value of the custom property as a string.

➤ **num_verify**, which compares the value of the custom property as a number.

➤ **bool_verify**, which compares the value of the custom property as a boolean expression.

You use a standard function whenever a simple comparison of property values can verify a custom property. For example, suppose you are checking the custom property *new_property*, and that the value is always a string such as "Object1" or "Object2". You can use the standard **string_verify** function to verify the *new_property* property.

Developing a Custom Verification Function

If your custom property requires complex comparison for verification, then the standard verification functions are inadequate, and you must develop your own custom verification function.

The verification function has the following prototype:

BOOL **verify_func** (HWND *hWnd*, LPSTR *str*);

Parameters

hWnd　　　　　The handle of the window or GUI object for which the property is being verified.

str　　　　　A buffer containing the value of the custom property. The value is established by the query function. The verification function compares this value to the required value of the custom property.

The verification function must return TRUE if the check passes, and FALSE if the check fails.

As with the query function, the verification function is written in C-language, and is compiled and linked into the same DLL as the query function.

For an example of a verification function for a custom property, see "Example of a Custom Property Function" on page 69.

Registering a Custom Property

In order to make a new custom property available to WinRunner, you must name the property and register it. You use the **add_record_attr** function to register a new custom property. This function has the following syntax:

add_record_attr (*attr_name*, *dll_name*, *query_func_name*, *verify_func_name* **);**

➤ *attr_name* is the name of the custom property that is being registered.

➤ *dll_name* is the full path and filename of the DLL in which the query and verification functions are defined.

➤ *query_func_name* is the name of the query function that is included in the DLL.

➤ *verify_func_name* is one of WinRunner's standard property verification functions, or the name of the verification function that is included in the DLL.

The following example registers the custom *new_property* property.

add_record_attr("new_property", "c:\\arch\\vb_util.dll", "new_property_query", "string_verify");

After executing the **add_record_attr** function, the new custom property, *new_property*, appears in the Available Properties list in the Configure Class dialog box. The new property is available to all standard and custom GUI object classes—you must assign it to the appropriate classes.
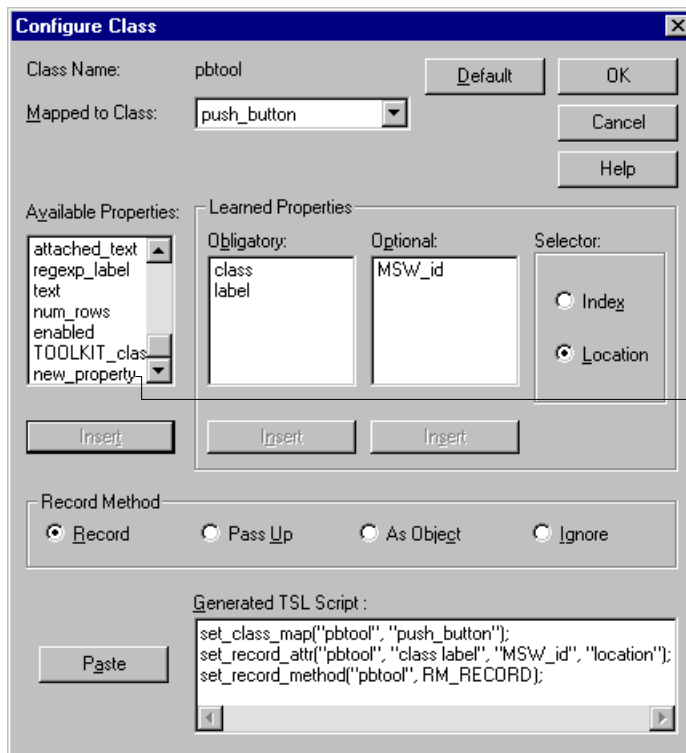
*The new custom property, **new_property**, was added using the **add_record_attr** function.*

Assigning a Custom Property to a GUI Object Class

After registering a custom property, you must assign the property to the appropriate custom class, usually as an obligatory property. You assign a property using the **set_record_attr** function. It is recommended that you use the Configure Class dialog box to generate the required **set_record_attr** function, and paste it into your test script.

The following example assigns the custom *new_property* property to the custom *cust_object* class.

set_record_attr("cust_object","class, new_property","MSW_id","location");

For additional information about the **set_record_attr** function, refer to the *TSL Online Reference*.

To assign a custom property to a GUI object class using the Configure Class dialog box:

1 Click **Tools** > **GUI Map Configuration**. The GUI Map Configuration dialog box opens.
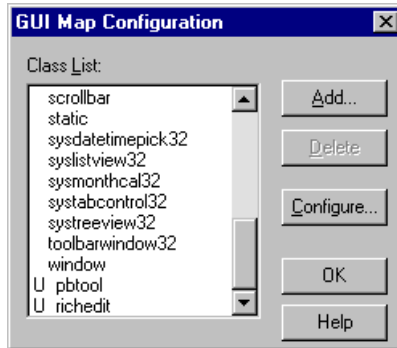
2 From the **Class List**, highlight the class with which you will associate the new custom property. Because your class is a custom class, you will find it at the bottom of the list, marked with a U (for user-defined) against the left border of the list.

3 Click the **Configure** button. The Configure Class dialog box opens.
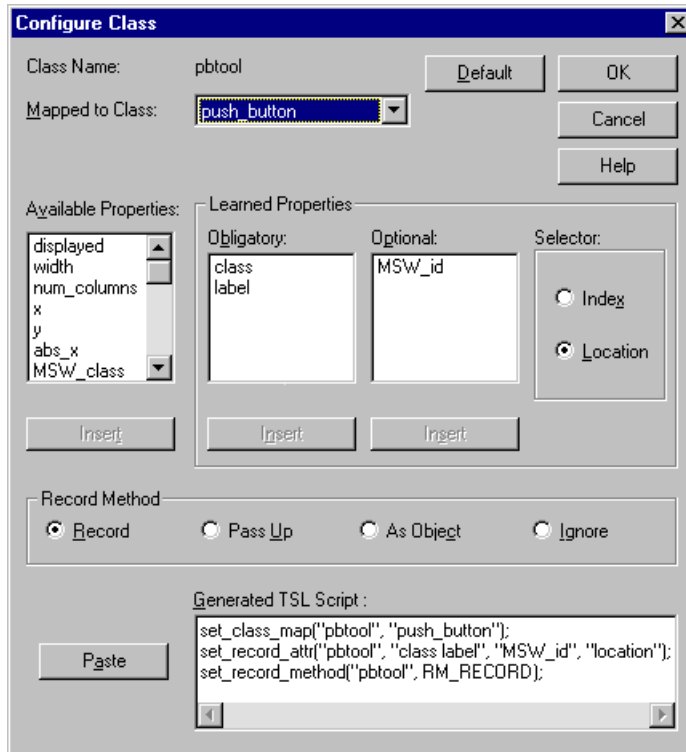
4 Locate and select your new custom property in the **Available Properties** list.

5 Click the **Insert** button below the list of obligatory properties. The new custom property is added to the list of obligatory properties. Examine the **set_record_attr** function in the **Generated TSL Script** box. Note that the new property is added to the list of obligatory properties.

6 Locate and select any property in the obligatory list which WinRunner no longer requires for unique identification.

7 Click the **Insert** button below the list of optional properties or below the list of available properties, as required. The selected property moves to the appropriate list.

8 Repeat steps 6 and 7 to remove all the obligatory properties that are no longer required.

Note: Although the class property may not be needed for unique identification, do not remove it from the list of obligatory properties.

---

9 Click **Paste** to paste the generated TSL statements into your test script.

10 Click **OK** to close the Configure Class dialog box.

Example of a Custom Property Function

The example below illustrates the structure and implementation of a custom property function. You may want to base the implementation of your custom property functions on this example.

In the example, the property query function is called *vb_name_query*. The *vb_name* property is equivalent to the "Name" property for controls in a Visual Basic application.

The value of the property is retrieved by calling the VBAPI function, VBGetControlName. VBGetControlName must be called on the stack of the task that owns the window "hWnd". This is done by means of window 'subclassing'. The window procedure of "hWnd" via SetWindowLong is changed to "VbCtrlProc". Then a (special) message is sent to "hWnd". This message is picked up by VbCtrlProc in the context of the task that created 'hWnd'. VbCtrlProc processes the message by calling VBGetControlName. The *cust_att.h* file sets the return values for the custom property function.

Filename: cust_att.h

```
#define WR_VB_SERVICE_STRING "MY_MESSAGE_IDENTIFIER_1"
enum {
    VB_GET_CTRL_NAME_IND,
    VB_GET_CTRL_INDEX_IND,
};

typedef struct {
    HWND hWnd;
    LPSTR value;
} VB_GET_CTRL_NAME_STRUCT;

typedef VB_GET_CTRL_NAME_STRUCT FAR* LPVBGCNS;
```

Filename: cust_att.c

```
#define EXPORTED _far _pascal __export

#include <windows.h>

#include <vbapi.h> // for VBGetHwndControl and VBGetControlName

#include "cust_att.h"

LRESULT EXPORTED VbCtrlProc(HWND hWnd,UINT message,WPARAM
wParam,LPARAM lParam);

HANDLE hmodDLL;

static UINT WR_VB_SERVICE_MSG = 0;

static FARPROC def_proc;

int FAR PASCAL LibMain
(
 HANDLE hModule,
 WORD wDataSeg,
 WORD cbHeapSize,
 LPSTR lpszCmdLine
)
{
```

```
 hmodDLL = hModule;
    WR_VB_SERVICE_MSG =
RegisterWindowMessage(WR_VB_SERVICE_STRING);
 return 1;
}

void EXPORTED vb_name_query(HWND hWnd, LPSTR value, int length)
{
    VB_GET_CTRL_NAME_STRUCT name_struct = {hWnd, value};

    def_proc = (FARPROC)SetWindowLong(hWnd, GWL_WNDPROC,
(LONG)VbCtrlProc);
    SendMessage(hWnd, WR_VB_SERVICE_MSG,
(WPARAM)VB_GET_CTRL_NAME_IND,
(LPARAM)(LPVBGCNS)&name_struct);
    SetWindowLong(hWnd, GWL_WNDPROC,(LONG)def_proc);
}

LRESULT EXPORTED VbCtrlProc(
    HWND hWnd,
    UINT message,
    WPARAM wParam,
    LPARAM lParam
)
{

    if(message == WR_VB_SERVICE_MSG) {
        switch(wParam) {
            case VB_GET_CTRL_NAME_IND:
            {
                LPVBGCNS p_name_struct = (LPVBGCNS)lParam;
                HCTL hctl = VBGetHwndControl(p_name_struct->hWnd);
                if(hctl)
                    VBGetControlName(hctl,p_name_struct->value);
                return((LRESULT)TRUE);
            }
            default:
                return((LRESULT)TRUE);
        }
```

```
        }
        return(CallWindowProc(def_proc, hWnd, message, wParam, lParam));
    }

    int EXPORTED vb_tag_query(HWND win, LPSTR value, int len)
    {
        vb_name_query(win, value, len);
        if(value[0] == '\0')
            return(FALSE);
        return(TRUE);
    }
```

7

Customizing Assigned Logical Names

You can customize the way that WinRunner assigns logical names to custom GUI objects.

This chapter describes:

➤ Understanding Logical Name Functions

➤ Developing a Logical Name Function

➤ Associating a Logical Name Function with a Custom GUI Object Class

About Customizing Assigned Logical Names

By implementing logical name functions, you can control the logical names that WinRunner assigns to custom GUI objects in your application. For example, suppose that your application contains a spinbutton that does not belong to any of WinRunner's standard GUI object classes. The spinbutton operates a counter for the quantity of tickets sold in a theater. When you click the spinbutton, WinRunner records a statement similar to the following:

obj_mouse_click ("SpinButton_2", 150, 300, LEFT);

The logical name, SpinButton_2, provides little indication of which GUI object is being acted on, especially if your application contains numerous spinbuttons. By implementing a logical name function you can improve the descriptiveness of the logical name that WinRunner assigns to the counter. In place of "SpinButton_2" your logical name function could return the logical name "Tickets_Sold_(spin)", resulting in the following recorded statement:

obj_mouse_click ("Tickets_Sold_(spin)", 150, 300, LEFT);

From the customized logical name, it is immediately apparent to which GUI object the recorded statement is referring.

To implement a logical name function, you perform the following steps:

1 Develop a logical name function that generates logical names for a custom GUI object class.

2 Associate the logical name function with a custom GUI object class.

Note that although you can modify the logical name of any GUI object using the GUI Map Editor:

➤ You can use the GUI Map Editor only *after* WinRunner has already assigned a logical name.

➤ When using the GUI Map Editor, you must *manually* modify *each* logical name, as required.

In contrast, a logical name function actually generates logical names the first time that WinRunner requires them, and a single logical name function can generate logical names for all the GUI objects in a given custom class.

You can combine a custom record function with a logical name function for a custom GUI object class. This combination ensures that the statements that WinRunner records into your test scripts are intuitive. That is, that the recorded statements describe both the actions performed, and the objects being acted upon. For more information, see Chapter 5, "Customizing Recorded Statements."

Understanding Logical Name Functions

When a GUI object is added to the GUI map, or when you use the GUI Spy to view the properties associated with a GUI object, WinRunner attempts to assign a descriptive logical name to the object. Therefore, if a GUI object has associated text, such as the label on a push button, WinRunner uses this text as the basis of the logical name. In most instances, this produces a logical name which is sufficiently descriptive of the GUI object.

If a GUI object has no associated text, WinRunner uses the *MSWclass* property as the basis of the logical name. This can result in a logical name that is not descriptive of the object which it represents. This, in turn, results in recorded test script statements that do not clearly describe the GUI objects with which they are associated. By implementing logical name functions, you enable WinRunner to establish and record more intuitive logical names for your custom GUI objects.

The logical names functions that you implement can employ various approaches to generate the required descriptive logical names. The approach you choose depends on the application being developed and the development environment used. The simplest approach is provided by those programmers who create databases containing descriptive details of all the GUI objects used in an application, as they develop the application. If you can find and access such a database, it may be the ideal source for all your custom logical names.

Developing a Logical Name Function

You write the logical name function in C-language format, and compile and link it into a DLL. You supply the name and location of the DLL when you associate the logical name function with a GUI object class. For more information, see "Associating a Logical Name Function with a Custom GUI Object Class" on page 76.

The logical name function has the following prototype:

int **function** (HWND *hWnd*, char* *str*, int *size*);

hWnd	The handle of the GUI object for which a logical name is being established.
str	A buffer allocated by WinRunner for the logical name. The logical name function stores the generated logical name in the *str* parameter.
size	The size of the *str* buffer, in bytes.

For an example of a logical name function for a custom object, see the example at the end of Chapter 6, "Adding Custom Properties for GUI Objects."

Associating a Logical Name Function with a Custom GUI Object Class

You use the **add_cust_record_class** function to associate a logical name function with a GUI object class. The **add_cust_record_class** function has the following syntax:

add_cust_record_class (*MSW_class*, *dll_name* **[,** *rec_func* **]**
   [, *log_name_func* **]);**

➤ *MSW_class* is the custom class with which the logical name function is associated. The custom class must have already been defined.

➤ *dll_name* is the full path and filename of the DLL in which you included the logical name function. If a custom record function exists for the GUI object class, then that function will be included in the same DLL. For more information, see Chapter 5, "Customizing Recorded Statements."

➤ *rec_func* is the name of the record function (if it exists) that is included in the DLL. The record function returns the string that will be recorded into the test script. For more information, see Chapter 5, "Customizing Recorded Statements."

➤ *log_name_func* is the name of the logical name function that you included in the DLL. The *log_name_func* function supplies custom logical names for custom GUI objects in the *MSW_class* class.

In the following example, the **add_cust_record_class** function associates the logical name function, vb_log_name, with the custom SpinButton class.

```
add_cust_record_class ("SpinButton", "c:\\winrun\\arch\\vb_util.dll", " ",
    "vb_log_name" );
```

Combining a Custom Record Function with a Logical Name Function

You can further improve the readability of your test scripts by implementing a custom record function together with a logical name function. You use the **add_cust_record_class** function to combine a custom record function with a logical name function for a custom GUI object class. This combination ensures that the statements that WinRunner records into your test scripts describe both the actions performed, and the objects being acted upon. For more information, see Chapter 5, "Customizing Recorded Statements."

For more information and an example of a logical name function combined with a custom property function, see Chapter 6, "Adding Custom Properties for GUI Objects."

Part III

Using the WinRunner API

8

The Mercury API Functions

This chapter contains all of the WinRunner API (Application Programmer Interface) functions that you need. This chapter includes:

➤ Type definitions for API functions

➤ Macros

The functions appear in the above order. Within the macro group, functions are arranged alphabetically. Some of the functions return or refer to WinRunner error codes. For more information on error codes, see the *TSL Online Reference*.

About API Functions

There are several types of functions in the API.

A *type* declaration defines how the execution functions you implement in the application are structured. The only type definition you need for inside testing is **MicFunctionProc**.

Macros can be used in the functions you implement in the application. The macros are:

➤ **mic_cp_from_string**

➤ **mic_cp_to_string**

➤ **mic_destroy_buf**

➤ **mic_destroy_string**

➤ **mic_get_object**

➤ **mic_init_buf**

➤ **mic_init_string**

➤ **mic_set_object**

You use the following types in macros:

➤ MicString

➤ MicObjectBuf

➤ MicObject

MicFunctionProc

Describes the prototype of any playback function (function type).

typedef int (MIC_PROTOTYPE ***MicFunctionProc**)(MicArgList *args*);

Registered with mic_if by **MicRegisterFunction.**

Argument

args	MicArgList is a pointer to a structure containing all of the parameters that are passed to the function.

Description

MicFunctionProc is the type for any playback function implemented by the toolkit that corresponds to a TSL playback command.(The toolkit is the development environment used to develop the application you are testing.) The function can have any number of input/output parameters. The parameters are passed in a data structure. In order to retrieve the parameters, the toolkit must use **MicExtractArgs.**

Return Value

This function returns MIC_E_OK for success or one of the WinRunner error codes for failure. For a list of error codes, refer to the *TSL Online Reference*.

mic_cp_from_string

Copies a MicString to a string (macro).

#include <mic_if.h>

mic_cp_from_string (*str*, *str_name*);

Parameters

str	A string to which MicString is copied.
str_name	The MicString.

Description

This macro copies a MicString to a string.

mic_cp_to_string

Copies a string to a MicString (macro).

#include <mic_if.h>

mic_cp_to_string (*str_name*, *str*);

Parameters

str_name	A MicString.
str	The string to be copied to the MicString.

Description

This macro copies a string to a previously initialized MicString.

mic_destroy_buf

Frees an object buffer of type MicObjectBuf (macro).

#include <mic_if.h>

mic_destroy_buf (*buf_name*);

Parameters

> *buf_name*　　　　　A buffer to be freed.

Description

> This macro frees an object buffer.

mic_destroy_string

Frees a string of type MicString (macro).

#include <mic_if.h>

mic_destroy_string (*str_name*);

Parameters

> *str_name*　　　　　A string to be freed.

Description

> This macro frees a string.

mic_get_object

Returns an object from an object buffer of type MicObjectBuf (macro).

#include <mic_if.h>

mic_get_object (*buf_name*, *index*);

Parameters

buf_name	The buffer from which the GUI object is retrieved.
index	The zero-based index of the GUI object in the buffer.

Description

This macro returns a GUI object from zero-based location *index* in an object buffer.

mic_init_buf

Initializes a buffer of type MicObjectBuf (macro).

#include <mic_if.h>

mic_init_buf (*buf_name*);

Parameters

buf_name	A buffer to initialize.

Description

This macro allocates a buffer named *buf_name*. To manage the buffer, you can use the following additional macros: mic_set_object, mic_get_object, mic_destroy_buf.

mic_init_string

Initializes a string of type MicString (macro).

#include <mic_if.h>

mic_init_string (*str_name*);

Parameters

str_name	The string to be initialized.

Description

This macro allocates a string. To manage the string, you can use the following additional macros: mic_cp_to_string, mic_cp_from_string, mic_destroy_string.

mic_set_object

Sets an object of type MicObject in an object buffer of type MicObjectBuf (macro).

#include <mic_if.h>

mic_set_object (*buf_name, index, object*);

Parameters

buf_name	The buffer in which the object is filled.
index	The index of the object in the buffer.
object	The object to be set.

Description

This macro fills an object at zero-based location *index* in an object buffer. To manage the buffer, you can use the following additional macros: mic_set_object, mic_get_object, mic_destroy_buf.

Index