



SilkTest® 2009 R2

Extension Kit for .NET

Borland®
(A MICRO FOCUS COMPANY)

**MICRO
FOCUS®**
Leading the Evolution™

Borland Software Corporation
4 Hutton Centre Dr., Suite 900
Santa Ana, CA 92707

Copyright 2009 Micro Focus (IP) Limited. All Rights Reserved. SilkTest contains derivative works of Borland Software Corporation, Copyright 2005-2009 Borland Software Corporation (a Micro Focus company).

MICRO FOCUS and the Micro Focus logo, among others, are trademarks or registered trademarks of Micro Focus (IP) Limited or its subsidiaries or affiliated companies in the United States, United Kingdom and other countries.

BORLAND, the Borland logo and SilkTest are trademarks or registered trademarks of Borland Software Corporation or its subsidiaries or affiliated companies in the United States, United Kingdom and other countries.

All other marks are the property of their respective owners.

November 2009

Contents

Introduction	5	QAP_ReleaseMouse	68
Who should read this manual	5	QAP_ReturnBoolean	70
Additional documentation	6	QAP_ReturnInteger	71
Typographical conventions	7	QAP_ReturnListClose	71
What this manual covers	8	QAP_ReturnListOpen	72
Overview	8	QAP_ReturnNull	73
Ease of use	9	QAP_ReturnReal	73
Power	9	QAP_ReturnString	74
Benefits	10	QAP_RouteAllClassFun	76
When to use the Extension Kit	10	QAP_RouteAllWindowsFun	77
		QAP_Terminate	79
Chapter 1 Tutorial	13	QAP_TypeKeys	80
Planning the extension	13	QAP_UnregisterClassFun	82
Writing a 4Test include file	15	QAP_UnregisterWindowFun	84
Understanding the UltraWinToolbar project	16		
		Chapter 5 Macro Reference	87
Chapter 2 Using the Extension Kit	19	GetArg	88
Overview	19	GetArgOpt	89
The framework of an extension function	21	GetArgType	90
Writing the function prototypes	25	GetListItem	90
Writing the extension function	26	IsArgNull	91
Driving the application under test	41	T_IsList	92
		T_IsNull	92
Chapter 3 Advanced Topics	45		
Internal extensions	45	Index	93
Graphical objects	45		
Internal functionality and data	50		
Client/server testing	50		
Functions that contain one or more arguments	50		
Chapter 4 Function Reference	53		
QAP_ClickMouse	54		
QAP_Initialize	56		
QAP_MoveMouse	57		
QAP_PressKeys	58		
QAP_PressMouse	60		
QAP_RaiseError	61		
QAP_RegisterClassFun	63		
QAP_RegisterWindowFun	65		
QAP_ReleaseKeys	67		

Introduction

About this manual

This manual provides all the information you need to work effectively with the SilkTest Extension Kit for .NET.

This chapter contains the following sections:

Section	Page
Who should read this manual	1
Additional documentation	2
Typographical conventions	3
What this manual covers	4
Overview	4
Ease of use	5
Power	5
Benefits	6
When to use the Extension Kit	6

Who should read this manual

This documentation assumes that you are a programmer who knows how to use SilkTest and who is familiar with C or C++, your compiler's operation, and the Windows API. Our example provides a project file for Microsoft Visual Studio .NET for Windows, and Microsoft Visual C++ for Windows.

Additional documentation

Information about SilkTest is provided in a multi-volume documentation set, as described in the following table. Updated versions of these documents are available online by clicking **Start/Programs/Silk/SilkTest 2009 R2/Documentation**, provided you chose to install the documentation when you installed SilkTest.

This Document...	Contains...
<i>SilkTest Online Help</i>	Concepts and procedures that describe how to use SilkTest to create testplans, perform cross-platform testing, customize tests, and use third-party GUI extensions. Also contains the 4Test Language Reference and descriptions of SilkTest dialogs. To access the Help, open SilkTest and click Help/Help Topics .
<i>SilkTest Installation Guide</i>	Instructions for installing SilkMeter on your license server and installing SilkTest on your local machine. Available from Start/Programs/Silk/SilkTest 2009 R2/Documentation/SilkTest Installation Guide .

Typographical conventions

Convention	Example	Description
<i>Italic</i>	<i>A script file is</i>	First use of a new term.
Courier Italic	Enter <i>install_dir</i>	Variable user input.
Courier Bold	Enter myscript.t	Actual user input, screen output, examples.
/	Record/Testcase	Indicates a menu pick, as in select the Testcase menu item from the Record menu.
Bold	ExceptPrint ()	Keywords or required punctuation. Leave exactly as shown.
Brackets []	SYS_Kill (iPid [, iSignal])	Optional arguments.
Ellipses (...)	access share-var [, share-var]... statements	Follows an element that can optionally be repeated.
Indentation	access share-var [, share-var]... statements	Indicates continuation of the statement on multiple lines indented another level.

What this manual covers

Chapter	Provides information about
Introduction	The power, ease of use, and benefits of the Extension Kit, and explains when you should use it.
Chapter 1, “Tutorial”	How to write a simple extension.
Chapter 2, “Using the Extension Kit”	Using the Extension Kit to write an extension.
Chapter 3, “Advanced Topics”	Writing external extensions, writing extensions so that SilkTest can interact with graphical (drawn) objects, writing functions for testing back-end communications for both the client and server, and loading libraries at runtime.
Chapter 4, “Function Reference”	Each Extension Kit function: a brief description, the syntax, the return value, an example, and the names of related functions.
Chapter 5, “Macro Reference”	The macros provided to simplify the retrieval of argument types and values from the ARGV structure that is passed to each extension function.

Overview

The Extension Kit provides one simple, but very powerful capability: it allows new 4Test Agent functions to be written in C. Being able to write a 4Test Agent function in C means that:

- you can call your function from your 4Test scripts
- your function runs on the target machine
- your function can interact with the Agent

The Extension Kit provides nearly limitless testing power.

You can now write extensions that will process both single byte and unicode (multibyte) data. The Extension Kit for SilkTest currently ships with a Unicode Agent.

If your extension is...	the extension can process
Unicode	Unicode, ANSI

If your extension is...	the extension can process
Multibyte	Multibyte, ANSI
ANSI	ANSI

Ease of use

If you are already a C programmer, the Extension Kit will be extremely easy to use. You will only need to learn a small number of data structures and functions. These structures and functions are documented in this manual and examples are provided online. It will take a typical C programmer no more than two or three hours to master the Extension Kit.

Power

External functions written with the Extension Kit automatically inherit the following features from the SilkTest Agent.

Network communications

Extensions communicate with the Agent running on the same machine. The Agent, in turn, communicates with 4Test either on the same machine or over the network. This means that an external function written in C automatically becomes a remote procedure call available to 4Test.

Error handling

By calling the built-in Extension Kit function, `QAP_RaiseError`, an external function can tell 4Test to raise an error. This raised error has the same visibility as any other test error — it can be handled in the script, and the error message you provide will automatically appear in the test results.

Powerful data types

Each extension function receives a single C argument. This argument is a data structure that contains all the information needed for the C code to access any 4Test data type (including complex records) that the script wishes to pass to the extension. Similarly, the functions provided by the Extension Kit for building the 4Test return value allow any simple or arbitrarily complex data to be returned from the external function to the 4Test script.

Window information

The data passed to each external function always includes the window handle of the object. This means that any extension function you write can easily interact with your custom GUI objects using window messages.

Event generation

External functions can generate keystrokes, mouse moves, or mouse clicks by calling built-in Extension Kit functions.

Benefits

Back-end testing	Ultimately, a combination of GUI and back-end (internal interface) testing is required to thoroughly test an application. Using the Extension Kit, you can easily develop back-end tests that access your application's internal data and integrate them with your GUI tests in a powerful 4Test framework.
Client/server testing	GUI test tools can generally only test the client side of a client/server system. With the Extension Kit, you can test both sides: Develop a few external interface functions, run this extension along with an Agent on your server, and run a single 4Test script on a third machine to easily drive the client and server sides of a complex application in parallel, correctly synchronizing their activities.
Custom GUI objects	The Extension Kit makes it easy to develop external functions that interact with your custom GUI objects. If you also create a 4Test window class for your object, then declarations for instances of the object can be generated with the Record/Window Declarations menu item. Your new object's methods will appear in both the Library Browser and the "point-and-click" Record Windows Identifiers dialog, making it easy to use them in your scripts.
Applications with no GUI at all	Even for pure back-end testing, you can use the Extension Kit to enhance the already powerful capabilities of SilkTest to implement robust test frames.

When to use the Extension Kit

Use the Extension Kit when you need to provide custom GUI support for complex applications by writing custom functions, or by adding new classes or methods to existing classes.

The Extension Kit allows you to extend the already powerful arsenal of SilkTest features for customizing tests, including:

- Mapping custom classes to standard classes.
- Adding 4Test classes and methods.
- Calling DLLs.

For information on mapping classes and adding new classes, see the SilkTest *online Help*.

Extensions versus DLLs

On Windows, SilkTest allows you to call dynamic link libraries (DLLs). The ability to call DLLs provides some of the same functionality as the Extension Kit without requiring you to write any C code. Depending on your needs, DLL calling may be an acceptable solution.

However, the Extension Kit provides the following advantages over calling DLLs:

- Functions written with the Extension Kit can run significantly faster.
- All 4Test data types are supported, so there is no need to convert between 4Test and C data types.
- Extension Kit functions are really “methods” of a specific class.
- You can use the Agent’s event-generation capabilities.
- You can create functions that take optional arguments.
- You can create functions that raise 4Test exceptions.
- The Extension Kit is available on all platforms; DLL calling is not.

1

Tutorial

In this chapter you will write a simple extension for the Infragistics UltraWinToolbars application. The example in this chapter is for an internal extension; this means that all of your test code will be written inside a DLL which will be loaded into the application.

In this chapter

This chapter covers:

Topic	Page
Planning the extension	9
Writing a 4Test include file	11
Understanding the UltraWinToolbar project	12

Planning the extension

The Infragistics UltraWinToolbars application is not part of the SilkTest installer. You must download a trial version from <http://www.infragistics.com>. This tutorial supports any version of the UltraWinToolbars application. You may have to update the project file for your version.

This tutorial was built using Microsoft's 1.1 .NET Framework.

Procedure To support your version of the Ultrawin toolbar

- 1 Open the project file (infra_toolbar.vcproj) using a text editor. If you open it using Microsoft's Visual Studio .NET, you may receive an error message that 'Project <project> failed to open'. The project file is provided in the Extension Kit directory under `ekwin32\tutorial\dotnet\infra_toolbar`.

- 2 Update the references to the version of the Ultrawin toolbar. The references are contained in the <AssemblyReference> tags within the <References> tag, and refer to Infragistics DLLs:
- 3 Open the project in Microsoft's Visual Studio .NET and rebuild the .dll.

Infragistics UltraWinToolbars sample application

The Infragistics UltraWinToolbars sample application and the Extension Kit DLL must be built using the same version of Microsoft's .NET Framework. The Extension Kit DLL example was built using Microsoft's 1.1 .NET Framework, so you must make sure that the Infragistics UltraWinToolbars sample application is also built with the 1.1 version of the Framework. If the Framework versions differ, the LoadLibrary() call, which SilkTest uses to map the DLL module into the address space of the .NET application, will fail. See the "LoadLibrary" online help topic for more information.

The application contains multiple toolbars that contain three powerful components that enable you to build rich user interfaces in your applications. The UltraWinToolbars component is designed to display and manage Microsoft Office style toolbars and menus. It also has an enhanced status bar component plus a dockable windows component which makes it easy to add a Microsoft Visual Studio.NET style dockable, tabbed windows interface to your application.

The Toolbar window contains multiple toolbars that contain different UIElements.

To familiarize yourself with the Infragistic UltraWinToolbars application:

- 1 Download the trial version of the UltraWinToolbars application from <http://www.infragistics.com>.
- 2 Start the UltraWinToolbars application.
- 3 Click some of the toolbar objects.

What to test

To test the application, you need to perform actions and retrieve information that a user might perform or retrieve manually. A few examples are:

- Click a toolbar button.
- Retrieve the names of the items in the specified toolbar.
- Retrieve the number of items in the toolbar.

**Add a method:
GetCount**

In this tutorial, you will write an extension to retrieve the number of items in the toolbar. This extension supports a new 4Test method that you will name GetCount. This method takes no arguments, and it returns an integer which is the number of objects in the toolbar.

Writing a 4Test include file

A complete extension consists of external functions written in C and declarations for the corresponding 4Test methods in a 4Test include file. The 4Test include file enables your 4Test scripts to call these external functions as methods to your custom class.

**Writing a window
class definition**

You need to write a window class definition for the UltraWinToolbar sample that contains the new class methods that your extension implements. The simplest way to define a window class is to derive it from an existing class.

The UltraWinToolbar window is similar to the existing CustomWin class, so you should derive your custom class from that.

Procedure To write a window class definition:

- 1 Start SilkTest (if it is not already running).
- 2 Open a new 4Test include file in the editor window.
- 3 Type the following into the include file:

```
winclass UltraWinToolbar : CustomWin
    extern integer GetCount()
```

This declaration defines a class called UltraWinToolbar, which is derived from the built-in 4Test class CustomWin. It inherits all of the methods and properties of the CustomWin class, plus any methods that you add.

The GetCount function is declared inside the window class definition using the extern keyword. The extern keyword indicates to 4Test that the function is implemented in an external extension rather than in a 4Test script.

- 4 Save this file as UltraWinToolbar.inc and minimize SilkTest.

Example

For a more advanced example of an extension for the UltraWinToolbar class that includes window declarations and a test case for the sample application, see the test script 'infratoolbar.t' that is provided in the ekwin32\tutorial\dotnet\infra_toolbar directory in your Extension Kit directory.

Understanding the UltraWinToolbar project

The “assist” library allows your extension to communicate with the SilkTest Agent. You need to link this library with your C or C++ project.

Preparing to write the extension

SilkTest comes with a header file that contains function prototypes for SilkTest functions, data structures that will be used by your extension functions, and some useful macros. To see the header file:

- 1 Open `segUltraToolbar.cpp` in the `ekwin32\tutorial\dotnet\infra_toolbar` directory in your Extension Kit directory.
- 2 The header file information is placed below the comment `/** include EK header files here */`. In this case, you see:

```
#include "qapwinek.h"
```

Preparing to build the extension

The “assist” library comes set up to communicate with the Agent and with include directories. To see how this is set up:

- 1 Open the project file using Microsoft’s Visual Studio .NET. The project file is provided in the Extension Kit directory under `ekwin32\tutorial\dotnet\infra_toolbar`.
- 2 Select **Project/Properties** from the main menu bar.
- 3 Select **Linker/Input** from the tree view in the Property Pages dialog and find the line “Additional Dependencies”. The `assist.lib` file is listed below that in the format:

```
"$(SEGUE_HOME)\ekwin32\assist.lib"
```

- 4 Select C/C++ from the tree view in the Property Pages dialog and find the line “Additional Include Directories”. You will see Infragistics’ include directory listed in the format:

```
"$(SEGUE_HOME)\ekwin32"
```

- 5 Save and close the project file.

2

Using the Extension Kit

The Extension Kit is commonly used to provide additional functionality for custom objects, but it can also be used to provide functionality for graphical objects, for back-end testing of client/server applications, and for testing internal functionality of an application.

This chapter describes the parts of an extension function, creating the framework surrounding extension functions, and the options available for driving the application under test (AUT).

In this chapter

This chapter covers:

Topic	Page
Overview	13
The framework of an extension function	15
Writing the function prototypes	19
Writing the extension function	20
Driving the application under test	35

Overview

Internal extensions This chapter explains the basics of creating an internal extension. The code of an internal extension is contained within a DLL that is loaded into the AUT's process by the 4Test, "LoadLibrary" function.

Overview of writing an internal extension

To write an internal extension for .NET, you must:

- 1 Make the assist.dll accessible to your AUT. See [“Making the assist.dll accessible” on page 15.](#)
- 2 Establish communication with the Agent. See [“Establishing communications with the Agent” on page 18.](#)
- 3 Register the extension functions. See [“Registering extension functions” on page 18.](#)
- 4 Write the function prototypes. See [“Writing the function prototypes” on page 19.](#)
- 5 Write the extension function. See [“Writing the extension function” on page 20.](#)

Parts of a SilkTest Extension

Extension Kit library The assist.dll library contains the functions described in this book. You must use the version of assist.dll that is shipped with the version of SilkTest that you are using. For information about linking assist.dll to your project, see [“Making the assist.dll accessible” on page 15.](#)

Header file The qapwinek.h header file contains the data structures, constants, and function prototypes described in this book.

The function and prototypes Extension functions are implemented in C or C++ and may be called from within a 4Test script.

- C function and prototype

Extension functions must be of a specified format. Each extension function automatically receives a standard data structure from SilkTest which contains arguments and other necessary information.
- 4Test prototype

SilkTest scripts that use extension functions must include 4Test prototypes for those functions. The prototype specifies the function's arguments and return type, and also contains a keyword that indicates that the function has been implemented in an extension.

Registration with the Agent The application that implements your extension functions must register with the SilkTest Agent upon its initialization.

Registration of functions Each extension function is a method of a SilkTest class or window, and must be registered with the SilkTest Agent. This registration specifies the name of the 4Test function, the name of the C or C++ function that implements it, the number and types of the function's arguments, and its return type.

Passing data from 4Test A pointer to a data structure containing 4Test arguments and relevant window information is automatically passed to all extension functions. Arguments are automatically converted from 4Test data types to corresponding C data types. The Extension Kit provides a set of macros which facilitate the retrieval of arguments from this structure.

Returning data to 4Test The Extension Kit provides a set of functions that convert values (including complex data structures such as records or multi-dimensional arrays) from C data types to 4Test data types, and return them to the script.

Raising exceptions The Extension Kit provides a function that will raise an exception in the 4Test script. The extension function may specify an error number and error message.

Driving the application Extension functions can manipulate the application under test by using the provided mouse and keyboard functions. Extension functions also have access to internal data and functionality.

The framework of an extension function

In addition to your C or C++ function, extensions require a specific framework and a series of initialization and registration procedures. This section covers:

- Making the assist.dll accessible
- Establishing communications with the Agent
- Registering extension functions
- Writing the function prototypes

Making the assist.dll accessible

In order to access the functionality of the Extension Kit, you must make the assist.dll accessible to your application under test (AUT), by linking it either implicitly or explicitly. Each of these options is described briefly in this section. For general information about linking .dlls, see http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vccore98/HTML/_core_determine_which_linking_method_to_use.asp.

Implicitly linking assist.dll

If you choose to implicitly link assist.dll to your AUT, you must do the following:

- reference `assist.lib` in your project file
- make sure the `assist.dll` can be found at runtime (note that you must either deploy `assist.dll` with your AUT, or change your DLL code to remove references to `assist.lib` and `assist.dll`)
- include the `qapwinek.h` header file in your extension

All of these files are located in your Extension Kit directory.

This option is easier to implement from a coding viewpoint.

Explicitly linking `assist.dll`

If you choose to explicitly link `assist.dll`, you must write code similar to that shown in the following C and C++ examples. Note that you do not need to reference `assist.lib` in your project/makefile or deploy `assist.dll` with your AUT because the execution of the Extension Kit code is conditional upon a successful load of `assist.dll`. You also do not need to include the `qapwinek.h` header file in your extension. While this option is a bit more difficult to code, it does not require you to change your DLL's code, or deploy `assist.dll`.

The following are examples of explicitly loading the `assist.dll` at runtime, using C and C++.

Sample C code

```
HINSTANCE hinstAssist;
FARPROC qap_init;

// Load the assist library
hinstAssist = LoadLibrary("assist.dll");

// Only try to register functions if the assist
// library loaded successfully
if (hinstAssist > HINSTANCE_ERROR)

// If you get a compiler error, try replacing
// the if line above with this one:
// if (hinstAssist > (HINSTANCE) HINSTANCE_ERROR)

{
    qap_registerClassFun = GetProcAddress(hinstAssist, \
        "QAP_RegisterClassFun");
    (*qap_registerClassFun)();

    /*** etc. ***/
}
```

The `LoadLibrary` function increases the reference count for the library every time it is called. Therefore, your application should call the `FreeLibrary` function when the library is no longer needed.

```

if (hinstAssist > HINSTANCE_ERROR)
{
    FreeLibrary(hinstAssist);
}

```

Sample C++ code

The following is an example of explicitly loading the assist.dll library on win32 platforms using C++. The example consists of two files, which are copied to the <SilkTest installation directory>/EKWIN32/Examples directory when you install the Extension Kit. These files are:

This file...	Contains...
FWxQapDynDll.h	The first half of this file is directly imported from the Extension Kit, which requires linking a 150 KB assist.lib that is the interface to a 2 MB assist.dll. This code creates a dynamic load interface. The second part of this file contains a C++ class definition to enable dynamic loading of the assist.dll directly.
FWxQapDynDll.cpp	The C++ source code.

Procedure Using the C++ Class Extension and Dynamic Loading for SilkTest’s Extension Kit

- 1 Add two files to your project: QapDynDll.cpp and QapDynDll.h.
- 2 Include QapDynDll.h in files referencing the EK facilities.
- 3 Instantiate an object, probably in the main or startup section of your application. It may be easiest to make it a class member of an “MyApp” object, a member pointer in an “MyApp” object or a global pointer created on start. For example:

```

main.h
class MyApp
{
public:
    QAPDynLoadEK* GetQapek() { return m_pQap; }
    .
    .
private:
    QAPDynLoadEK *m_pQap;// member pointer to QAP
    QAPDynLoadEK m_Qap;// member object of QAP
    .
    .
}

```

```

    }
    main.cpp
    (if using the member pointer, the following needs to be
     done)
    m_pQap = new QAPDynLoadEK();

```

- 4 After compiling your application, ensure the Agent is running and assist.dll is on your path.

Note It's probably easiest to put assist.dll in your \winnt or \windows directory.

- 5 Start your application and try a test example.

Note Many of the definitions in the wapwinek were removed or changed, as they are not needed or may clash with other definitions. Most of the basic EK and documentation remains roughly valid. Differences include substituting a '.' for the '_' in functions. For example:

QAP_RegisterWindowFun() becomes QAP.RegisterWindowFun()
(assuming QAP is the named identifier).

Establishing communications with the Agent

Your .NET EK DLL will automatically register with the SilkTest Agent when its loaded. This is done by a call to the 4Test LoadLibrary function. This function returns TRUE if it successfully loads the DLL with the .NET Extension or FALSE if it does not.

Note In order for your extension to successfully register with the SilkTest Agent, the Agent must be running before the extension is started.

For example:

```

Testcase LoadEkDll() appstate none
    bVal=SwfDialogBox("UltraToolbars").LoadLibrary("c:\UltraToolbar.dll")

```

Registering extension functions

In order for your extension functions to be accessible to SilkTest, you must register them with the SilkTest Agent.

Each function for a custom class or window must be registered with the Agent at the time of your DLL's initialization after the application itself has been registered. To register a function for a custom class, use the QAP_RegisterClassFun function.

The QAP_RegisterClassFun function takes as arguments:

- The name of the class.
- The 4Test method name.
- A pointer to the C extension function which implements the method.
- The 4Test return type.
- The number of 4Test arguments.
- Flags describing the attributes and types of each of the arguments.

Given the following 4Test function for the custom class MyClass:

```
winclass MyClass
    extern INTEGER Increment (INTEGER iNum,
                             out INTEGER iFinal)
```

The function might be registered as follows:

```
QAP_RegisterClassFun("MyClass", "Increment", \
    MC_Increment,
    T_INTEGER, 2,
    P_IN | T_INTEGER,
    P_OUT \
    | T_INTEGER);
```

For more information about data types and parameter attributes, see [“4Test data types” on page 21](#) and [“Parameter attributes” on page 22](#).

Writing the function prototypes

Writing the C or C++ prototype

All function prototypes for SilkTest extension functions have the following form:

```
void QAPFUNC FunctionName (PARGS pArgs);
```

All SilkTest extension functions are automatically passed a single PARGS argument, which contains the 4Test arguments and pertinent window information. For more information about retrieving arguments and returning values, see [“Retrieving 4Test arguments” on page 22](#) and [“Returning values to 4Test” on page 32](#).

Writing the 4Test prototype

In order to make your extension functions available to 4Test, you must write and compile an include file that contains all the necessary information about your custom class and its functions.

Your 4Test include file must contain a window class definition for your custom class which includes declarations of your extension functions.

For example, to define a custom class called MyText, which derives from the standard TextField class and implements two functions called SetText and GetText, your 4Test window class definition would read as follows:

```
winclass MyText : TextField
    extern void SetText (STRING s)
    extern STRING GetText ()
```

The extern keyword indicates to SilkTest that the function is not written in 4Test, but is implemented externally in an extension.

Writing the extension function

What the extension function does

The extension function itself may do the following things:

- Retrieve 4Test arguments.
- Return values to 4Test.
- Raise 4Test errors.
- Use mouse and keyboard events to manipulate the application under test.
- Access internal data within the application.
- Access internal functions in the application.
- Access functions available via libraries.

In this section

This section covers:

- 4Test data types
- Parameter attributes
- Retrieving 4Test arguments
 - The ARGS structure
 - Simple arguments
 - Optional arguments
 - Complex data structures
 - Returning values to 4Test
- Returning values to 4Test

- 4Test return values
- Out arguments
- Complex Data Structures
- Raising 4Test errors

4Test data types

4Test uses data types that are similar to, but not exactly the same as, C data types. The Extension Kit performs the conversion automatically for data passed into extension functions. It also provides a set of functions that perform the type conversion when returning data to a 4Test script. For more information about 4Test data types, see the *online Help*.

The header file for the Extension Kit defines constants for the built-in 4Test data types. Those constants and the 4Test data types they correspond to are as follows:

TYPE Value	4Test Data Type
T_VOID	void
T_NULL	null (any type in 4Test can be null)
T_UNSET	the variable hasn't been given an initial value yet
T_BOOLEAN	boolean
T_INTEGER	integer (equivalent to a C "long")
T_REAL	real
T_STRING	string or enumerated type
T_NUMBER	number
T_ANYTYPE	anytype
T_LIST_BOOLEAN	list of boolean
T_LIST_INTEGER	list of integer
T_LIST_REAL	list of real
T_LIST_STRING	list of string
T_LIST_NUMBER	list of number
T_LIST_ANYTYPE	list of anytype or record

Note Enumerated types are passed as strings in order to avoid a mismatch of numerical values between 4Test code and C code.

Parameter attributes

Each 4Test argument has a set of attributes indicating whether the argument is **in**, **out**, or **inout**; whether the argument may have a NULL value; and whether the argument is optional. The defaults are in, non-NULL, and non-optional. For more information about these attributes, see the *online Help*. The header file for the Extension Kit defines constants for each of these attributes, as shown in the table below:

Parameter Attribute	Description
P_IN	in
P_OUT	out
P_INOUT	inout
P_ALLOW_NULL	allow parameter to be NULL
P_OPTIONAL	parameter is optional

Retrieving 4Test arguments

The ARGS structure Each of your extension functions takes a single argument of type PARGS, which is a pointer to an ARGS structure. This structure is passed to your function automatically from SilkTest and contains all necessary information about the window to be accessed and the arguments given to the 4Test function.

The ARGS structure is defined as follows:

```
typedef struct
{
    HWND hWnd;
    Window window;
    Widget widget;
    INT int iCount;
    PDATA pData;
} ARGS, FAR *PARGS;
```

The ARGS structure contains the following elements:

Element	Explanation
HWnd	A handle to the window for which the function was called.
ICount	The number of arguments passed to the function.
PData	A pointer to a structure containing the values and types of the arguments.

The element pData is a pointer to a structure of type DATA, which contains the information about the arguments that are passed to the 4Test function. The DATA structure is defined as follows:

```
typedef struct _DATA
{
    TYPE Type;
    TYPE Reserved;
    union
    {
        BOOL fValue;
        LONG lValue;
        double dblValue;
        PSZ LPSTR char *pszValue;
        struct _LIST
        {
            int iCount;
            struct _DATA FAR *pData;
        } List;
    } Value;
} DATA, FAR *PDATA;

typedef struct _LIST LIST, FAR *PLIST;
```

The DATA structure contains these elements:

Element	Explanation
Type	The 4Test data type.
Value	A union of values of each possible type.

The Type element is a 4Test data type, as defined in [“4Test data types” on page 21](#).

The Value element is a union of elements corresponding to each possible 4Test type. Because your extension functions are declared in your 4Test include file (see [“Writing the function prototypes” on page 19](#)), SilkTest performs type checking on the arguments to your function so that you can be assured that the correct Type value has been set. Note that this union includes another PDATA, making the structure recursive. This allows for the more complex 4Test data types, such as records and nested lists, to be passed to C extensions.

Simple arguments The Extension Kit provides a macro called GetArg which simplifies the syntax for accessing the values of arguments from the ARGS structure.

```
#define GetArg(num, type) (pArgs->pData[num].Value.type)
```

For example, the following 4Test function takes an integer as an argument, adds 1 to it, and returns that value. Your 4Test include file would contain:

```
winclass MyClass
    extern INTEGER Increment (INTEGER i)
```

Your extension function will be passed PARGS, and it will need to extract the value of the first argument, increment it by 1, and return it. The following code sets the variable *i* to the value of the first argument in the 4Test function, increments it, and returns that value to 4Test:

```
void QAPFUNC Increment(PARGS pArgs)
{
    long i;

    i = GetArg(0, lValue);
    i++;

    QAP_ReturnInteger (RETVAL, i);
}
```

Optional arguments 4Test allows arguments to be marked as optional. If the user does not provide a value for an optional argument, the value passed to your function will be of type NULL. Your function should provide a default value for any optional arguments.

The Extension Kit provides a macro called `GetArgOpt` which simplifies the syntax for accessing optional arguments from PARGS and setting their default values. This macro takes the argument number, argument type, and default value as its arguments. It returns either the value entered by the user, or the default value you have provided.

```
#define GetArgOpt(num, type, default) \
    (T_IsNull (pArgs->pData[num].Type) ? default
: \
    pArgs->pData[num].Value.type)
```

For example, the following 4Test function takes two integer arguments, the second of which is optional. The function increments the first argument by the value of the second, or by 1 if there is no second argument, and returns that value. Your 4Test include file would contain:

```
winclass MyClass
    extern integer Increment(integer iFirst,
        integer iSecond optional)
```

Your extension function will be passed PARGS, and it will need to extract the value of the first argument, extract the value of the second argument or set the default, increment the first value by the proper amount, and return that value. The following code sets the variable *i* to the value of the first argument in the 4Test function, sets the variable *j* to the value of the second argument or the default, increments *i*, and returns the new value to 4Test:

```
void QAPFUNC Increment(PARGS pArgs)
{
    long i, j;

    i = GetArg(0, lValue);
    j = GetArgOpt(1, lValue, 1);
    i = i + j;
    QAP_ReturnInteger(RETVAl, i);
}
```

Note If your function allows optional arguments, you must specify this when you register the function. For more information about registering functions, see [“Registering extension functions”](#) on page 18.

Complex data structures 4Test lists and records are, in essence, arrays. The Value element of the DATA structure contains an element called List, which is a structure containing a count of the number of elements in the list or record and a pointer to an array of DATA structures. These DATA structures, in turn, contain the values of the individual elements of the list.

Enumerated types Enumerated types are passed between 4Test and C as strings. This avoids the problem of inconsistencies between 4Test and C header files. For example:

```
LPSTR myEnum = GetArg(0, pszValue);
```

Lists The Extension Kit provides a macro called GetListItem which simplifies the syntax when accessing a list argument in PARGS. This macro takes a list, item number, and item type as arguments.

```
#define GetListItem(list, item, type) \
    ((list).pData[item].Value.type)
```

For example, assume a function called AddList, which takes a list of integers, adds them, and returns the total.

```
void QAPFUNC AddList(PARGS pArgs)
{
    long i, cnt;
```

```
LIST theList;

i = 0;
theList = GetArg(0, List);

for(cnt=0 ; cnt < theList.iCount ; cnt++)
{
    i += GetListItem(theList, cnt, lValue);
}

QAP_ReturnInteger(RETVAl, i);
}
```

Values of nested lists are retrieved in the same manner. For example, the following function takes as an argument a list of list of integer (that is, a nested list), adds the elements of the secondary lists, then adds the subtotals. It then returns the final result:

```
void QAPFUNC AddList(PARGS pArgs)
{
    long i, j, cnt1, cnt2;
    LIST list1, list2;

    i = 0;
    list1 = GetArg(0, List);

    for(cnt1=0 ; cnt1 < list1.iCount ; cnt1++)
    {
        j = 0;
        list2 = GetListItem(theList, cnt1, List);

        for(cnt2=0 ; cnt2 < list2.iCount ; cnt2++)
        {
            j += GetListItem(list2, cnt2, lValue);
        }

        i += j;
    }

    QAP_ReturnInteger(RETVAl, i);
}
```

4Test records have the same internal structure as lists of anytype, and are retrieved in the same manner. For example, the function below takes the following 4Test record as an argument:

```
type PERSON is record
    STRING sName
    INTEGER iAge
```

The 4Test prototype is:

```
winclass MyClass
    extern BOOLEAN IsAdult (PERSON Person)
```

The function is registered as follows:

```
QAP_RegisterClassFun("MyClass",
    "IsAdult",
    MC_IsAdult,
    T_BOOLEAN,
    1,
    P_IN | T_LIST_ANYTYPE);
```

The function returns TRUE if the person is 18 years or older, and FALSE otherwise:

```
void QAPFUNC MC_IsAdult(PARGS pArgs)
{
    long age;
    LIST person;

    person = GetArg(0, List);

    age = GetListItem(theList, 1, lValue);

    if (age >= 18)
        QAP_ReturnBoolean(RETV, TRUE);
    else
        QAP_ReturnBoolean(RETV, FALSE);
}
```

Passing lists with nested records The following is an example for passing lists with nested records. The example shows three files:

- ek1inc (below on this page) - the include file defining the objects and structures that the ek1.t test script needs in order to run
- ek1.t (on page 29) - the test script file
- A portion of C/C++ code file that is actually part of the extension (also on page 30)

The include file, ek1.inc:

```
//-----
// ek1.inc
```

2 USING THE EXTENSION KIT

Writing the extension function

```
// test to work out data structures needed

type SType is enum

// change type to const so doesn't change to string on
// return

{
    Line = 1,
    Circle = 2,
    Ellipse = 3,
    Arc = 4,
    Rect = 5
}
type SketchObject is record
{
    SType    kind;
    int      ObjNum;
    POINT    p1;
    POINT    p2;
    POINT    p3;
    int      numconst;
    int      stconst;
}
type CType is enum
{
    Perp = 1,
    Parallel = 2,
    Coincident = 3,
    Horizontal = 4,
    Vertical = 5,
    Tangent = 6
}
type Constraint is record
{
    CType    ctype; // the type of relationship
                //between the objects
    int      o1; // first object in the constraint
    int      o2; // 2nd object - typically blank
    POINT    p1; // coordinate of constraint point
    POINT    p2; // 2nd coord of constraint point -
                //typically blank
}
window MainWin Ek1
{
    tag "$H:\VCprojs\QAPtest\ek1\Debug\ek1.exe";

    Menu File
    {
        tag "File";

        MenuItem Exit
        {
            tag "$105";
        }
    }
}
```



```

    }
}

Menu Help
{
    tag "Help";
MenuItem About
{
    tag "$104";
}
}

// returns the number of objects, an array of the
//objects, an array of constraints

extern INT MSketchGetAll (out LIST OF SketchObject sobjs, out LIST OF Constraint
                        constraints);
extern INT Mtest (out ANYTYPE p, out ANYTYPE q);
extern INT Mtest0 (out INTEGER a, out INTEGER b);
}
//-----

```

The test script, ek1.t:

```

//ek1.t
use "ek1.inc";

main()
{
LIST of SketchObject sobjs;
LIST of Constraint      constraints;
int num, a,b;
ANYTYPE p, q;

    Ek1.SetActive ();
    Ek1.Help.About.Pick ();
    Ek1.DialogBox("About").SetActive ();
    Ek1.DialogBox("About").PushButton("OK").Click(); Ek1.SetActive ();
    Ek1.Move (40, 6);
    Ek1.Size (662, 298);

    num = Ek1.Mtest0(a, b);
    Print(num, a, b);

    num = Ek1.Mtest(p, q);
    Print(p);
    Print(q);

    num = Ek1.MSketchGetAll(sobjs, constraints);
    Print(num);
    Print(sobjs);
    Print(constraints);
}
//-----

```

The portion of C/C++ code that is part of the extension:

```
#include "stdafx.h"
#include "..\qapwinek.h"

BOOL fAgentRunning;

extern "C"
{

// Before trying to load this, you should build two
// arrays of data...objects and constraints

void QAPFUNC MSketchGetAll (PARGS pArgs)
{
int i;
// load the sketch objects using QAP's parameter
// passing
QAP_ReturnListOpen(0);
for (i=0; i < 10 ;++i)
{
QAP_ReturnListOpen(0);
QAP_ReturnInteger(0, 1); // type of
//object: line,arc...
QAP_ReturnInteger(0, i); // Object num

QAP_ReturnListOpen(0);
QAP_ReturnInteger(0, i*50); // point 1 - x
QAP_ReturnInteger(0, i*55); // point 1 - y
QAP_ReturnListClose(0);

QAP_ReturnListOpen(0);
QAP_ReturnInteger(0, i*60); // point 2 - x
QAP_ReturnInteger(0, i*65); // point 2 - y
QAP_ReturnListClose(0);

QAP_ReturnListOpen(0);
QAP_ReturnInteger(0, i*70); // point 3 - x
QAP_ReturnInteger(0, i*75); // point 3 - y
QAP_ReturnListClose(0);

QAP_ReturnInteger(0, i); // num const
QAP_ReturnInteger(0, i); // st const

QAP_ReturnListClose(0);
}
QAP_ReturnListClose(0);

// Now load the constraint
QAP_ReturnListOpen(1);
for (i=0; i < 10 ;++i)
{
QAP_ReturnListOpen(1);
```

```
    QAP_ReturnInteger(1, 2); // type of object:
    // line, arc...
    QAP_ReturnInteger(1, i); // Object num
    QAP_ReturnInteger(1, i); // Object num

    QAP_ReturnListOpen(1);
    QAP_ReturnInteger(1, i*50); // point 1 - x
    QAP_ReturnInteger(1, i*55); // point 1 - y
    QAP_ReturnListClose(1);

    QAP_ReturnListOpen(1);
    QAP_ReturnInteger(1, i*60); // point 2 - x
    QAP_ReturnInteger(1, i*65); // point 2 - y
    QAP_ReturnListClose(1);

    QAP_ReturnListClose(1);
    }
    QAP_ReturnListClose(1);

    QAP_ReturnInteger(RETV, 100);
}
void QAPFUNC test (PARGS pArgs)
{
    int i;

    // load the sketch objects using QAP's parameter
    // passing

    QAP_ReturnListOpen(0);
    QAP_ReturnListOpen(0);

    QAP_ReturnInteger(0, 99);
    QAP_ReturnInteger(0, 66);

    QAP_ReturnListClose(0);

    QAP_ReturnListOpen(0);

    QAP_ReturnInteger(0, 44);
    QAP_ReturnInteger(0, 55);

    QAP_ReturnListClose(0);
    QAP_ReturnListClose(0);

    QAP_ReturnListOpen(1);
    QAP_ReturnListOpen(1);

    QAP_ReturnInteger(1, 99);
    QAP_ReturnInteger(1, 66);

    QAP_ReturnListClose(1);

    QAP_ReturnListOpen(1);
```

2 USING THE EXTENSION KIT

Writing the extension function

```
        QAP_ReturnInteger(1, 44);
        QAP_ReturnInteger(1, 55);

        QAP_ReturnListClose(1);
        QAP_ReturnListClose(1);

        QAP_ReturnInteger(RETVAl, 10);
    }

void QAPFUNC test0 (PARGS pArgs)
{
    QAP_ReturnInteger(0, 77);    // type of object:
    // line, arc...
    QAP_ReturnInteger(1, 67);    // type of object:
    // line, arc...
    QAP_ReturnInteger(RETVAl, 10);
}
}

BOOL APIENTRY DllMain( HANDLE hModule,
    DWORD ul_reason_for_call, LPVOID lpReserved)
{
    switch (ul_reason_for_call)
    {
        case DLL_PROCESS_ATTACH:
            /*** REGISTER WITH THE SILKTEST AGENT ***/
            RegisterQAPFunctions();
            break;
        case DLL_THREAD_ATTACH:
        case DLL_THREAD_DETACH:
        case DLL_PROCESS_DETACH:
            break;
    }
    return TRUE;
}

void RegisterClassFunctions()
{
    got = QAP_RegisterWindowFun("Ek1", "Mtest0", test0,
        T_INTEGER, 2,
        P_OUT | T_INTEGER, P_OUT | T_INTEGER );
    got = QAP_RegisterWindowFun("Ek1", "Mtest", test,
    T_INTEGER, 2, P_OUT | T_LIST_ANYTYPE, P_OUT | T_LIST_ANYTYPE );
    got = QAP_RegisterWindowFun("Ek1", "MSketchGetAll",
    MSketchGetAll, T_INTEGER, 2, P_OUT | T_LIST_ANYTYPE,
    P_OUT | T_LIST_ANYTYPE);
}
}
```

Returning values to 4Test SilkTest extension functions have no return value in C. They return values to 4Test by means of the following set of function calls:

- QAP_ReturnNull

- QAP_ReturnBoolean
- QAP_ReturnInteger
- QAP_ReturnReal
- QAP_ReturnString

Each of these functions takes as parameters an argument number and, if applicable, the return value.

4Test return values The first argument to each of the Extension Kit's return functions is the number of the argument in which to return the value. The constant RETVAL indicates that the value should be returned as the return value of the corresponding 4Test function.

For example, the following 4Test function takes an integer as an argument, adds 1 to it, and returns that value. Your 4Test include file would contain:

```
winclass MyClass
    extern INTEGER Increment (INTEGER i)
```

The following example returns the resulting value to 4Test as the function's return value:

```
void QAPFUNC MC_Increment(PARGS pArgs);
{
    long i;

    i = GetArg(0, lValue);
    i++;

    QAP_ReturnInteger(RETVAL, i);
}
```

Out arguments The 4Test language allows arguments to be specified as **in**, **out**, or **inout**. Arguments are set to **in** by default.

For example, the following 4Test function takes an integer and an out integer as arguments, adds 1 to the first argument, and sets the second argument to that value. Your 4Test include file would contain:

```
winclass MyClass
    extern INTEGER Increment (INTEGER iNum,
        out INTEGER iFinal)
```

The following code returns the resulting value to 4Test as the second argument:

```
void QAPFUNC MC_Increment(PARGS pArgs);
{
```

```
        long i;  
  
        i = GetArg(0, lValue);  
        i++;  
  
        QAP_ReturnInteger(1, i);  
    }  
}
```

Complex Data Structures A list is returned using the QAP_ReturnListOpen and QAP_ReturnListClose functions in conjunction with the other QAP_Return* functions. The QAP_ReturnListOpen function marks the beginning of a list, and the QAP_ReturnListClose function marks the end of the list.

For example, the following 4Test function takes three integers as arguments and returns a list of integer with those arguments as elements. Your 4Test include file would contain:

```
winclass MyClass  
    extern LIST OF INTEGER MakeList (INTEGER i1,  
        INTEGER i2, INTEGER i3)
```

The following code returns the arguments within a list as the 4Test return value:

```
void QAPFUNC MC_MakeList(PARGS pArgs);  
{  
    long i, j, k;  
  
    i = GetArg(0, lValue);  
    j = GetArg(1, lValue);  
    k = GetArg(2, lValue);  
  
    QAP_ReturnListOpen(RETVAl);  
        QAP_ReturnInteger(RETVAl, i);  
        QAP_ReturnInteger(RETVAl, j);  
        QAP_ReturnInteger(RETVAl, k);  
    QAP_ReturnListClose(RETVAl);  
}
```

The QAP_ReturnListOpen and QAP_ReturnListClose functions may be nested in order to return nested lists. For instance, to return the following 4Test list:

```
    {{1, a}, {2, b}}
```

use the following:

```
    QAP_ReturnListOpen(RETVAl);
```

```
QAP_ReturnListOpen (RETVAl) ;  
    QAP_ReturnInteger (RETVAl, 1) ;  
    QAP_ReturnString (RETVAl, "a") ;  
QAP_ReturnListClose (RETVAl) ;  
QAP_ReturnListOpen (RETVAl) ;  
    QAP_ReturnInteger (RETVAl, 2) ;  
    QAP_ReturnString (RETVAl, "b") ;  
QAP_ReturnListClose (RETVAl) ;  
QAP_ReturnListClose (RETVAl) ;
```

Records have the same internal structure in 4Test as lists, and are returned in the same manner.

Raising 4Test errors

4Test uses the 4Test function prototype to automatically check the number and types of arguments and raise errors when appropriate. The Extension Kit also provides a function called QAP_RaiseError, which allows you to raise 4Test errors from within your extension.

QAP_RaiseError takes as arguments an error number, a printf-style error format string, and values for that string. For example, the following code will raise an error with error number 1 and the message "Error: the maximum value is 10":

```
int iMax = 10;  
QAP_RaiseError(1, "The maximum value is %d",  
iMax);
```

Driving the application under test

Driving versus observing

The functionality that your extension functions will implement can be broken into two broad categories: driving the application and observing the application's state.

The philosophy of SilkTest is to test the application, wherever possible, as a user would. While it is possible to drive an application to a desired state using internal function calls, you should do so through mouse and keyboard event-generation. In order to retrieve information about a custom object's state, it is usually necessary to use internal function calls.

Generating mouse and keyboard events

The Extension Kit allows you to use the SilkTest Agent to manipulate your application by sending keyboard and mouse events. Functions written with the Extension Kit will be faster than equivalent 4Test functions, and your extension will have the added advantage of having access to internal information about your application, such as the size and coordinates of GUI objects.

You should use the Extension Kit's mouse and keyboard functions to manipulate your application whenever possible. While your extension has access to internal functions within your application, keep in mind that your goal is to test the application as a user would. Therefore, using these mouse and keyboard functions will give you the most reliable results. The SilkTest Agent follows this principle.

The Extension Kit provides the following functions for sending mouse and keyboard events to your application:

- QAP_ClickMouse
- QAP_PressKeys
- QAP_ReleaseKeys
- QAP_PressMouse
- QAP_ReleaseMouse
- QAP_MoveMouse

These functions require the Mac Window Pointer (or NULL for the Desktop) and the handle (or NULL for a top-level window) of the handle of the window and the widget you want to manipulate and the keys you want to type or the coordinates at which to operate the mouse. A window pointer and handle are handle is and the widget are passed to your extension function as a part of PARGSthose or any others. You can determine the coordinates, relative to the window, by using information available in your application. See [“Accessing internal functionality and data” on page 37](#).

Flush, Focus, and DelayFocus Each of the above functions takes as an argument a set of flags, which set options for event generation, including the following:

- **Flush.** By default, no keyboard or mouse events are generated until a state is reached in which doing so would leave nothing pressed. This means that a series of mouse or key presses will not be sent until the corresponding mouse or key releases are also issued. You can override these defaults by using the EVT_FLUSH or EVT_NOFLUSH options.
- **Focus.** By default, keyboard events are sent without setting the keyboard focus. In order to set the focus to the specified object before generating the event, use the EVT_SETFOCUS flag.
- **Delay.** Within the Agent Options dialog, the user can set mouse and keyboard delays that will be used by the Agent for all 4Test methods. You can ignore these delays in your extension by using the EVT_NODELAY flag.

**Accessing internal
functionality and
data**

While Borland recommends that you use the Extension Kit's mouse and keyboard functions whenever possible, some functionality may not be available through those means. Even when the keyboard and mouse functions are appropriate, you may require internal information from your application in order to determine coordinates for the actions.

Your extension function has easy access to the internal functionality and data of your application. Since your extensions are internal, all global variables and data structures will be directly available.

2 USING THE EXTENSION KIT
Driving the application under test

3

Advanced Topics

In this chapter

This chapter covers:

Topic	Page
Internal extensions	39
Graphical objects	39
Internal functionality and data	44
Client/server testing	44
Functions that contain one or more arguments	44

Internal extensions

Your SilkTest extension functions can be written directly into your application by creating your own DLL and calling the `4Test LoadLibrary` method to load the DLL into the application's process.

Graphical objects

The Extension Kit's uses are not limited to custom objects which are recognized as `CustomWins` by SilkTest. The recording tools of SilkTest don't recognize graphical (drawn) objects as individual windows, but it's still possible to drive and retrieve information about those objects by means of extension functions.

SilkTest tags

SilkTest uses *tags* to uniquely identify objects in an application. Using the class names and tags in your declaration, SilkTest resolves identifier names into the unique identifiers that are used on the specific platform (window handles, window/widget pairs, control IDs, and so on). This allows SilkTest to specify the correct object when communicating with the application.

Graphical objects aren't really "objects" at all, but are implemented as drawings within a parent object. They don't have any sort of unique identifier, and therefore don't have a SilkTest tag.

Creating an identification system

Graphical objects are invisible to the SilkTest recorders, but it's possible to create your own system for uniquely identifying these objects. When a 4Test script calls an extension function, it must be able to indicate to the extension code which "object" the script is referring to. When developing an identification system, you can use anything you want to uniquely identify an object: an integer, a string, coordinates, or any other piece of information that makes sense for your custom object.

For instance, in the Infragistics sample application the toolbar objects are non-native windows and thus are not recognized by the SilkTest recording tools, but it's possible to treat each item as a separate object with the use of the Extension Kit. Given that the cells appear in a regular grid, an obvious choice for a unique identification system is the use of grid coordinates.

The "pseudotag" It's not possible to use the SilkTest tag as a holder for your identification information, because it won't resolve to a valid object. However, you can create your own variable in which to store this information. Although a tag must be a string, this "pseudotag" may be of any type. The examples\graphic\scripts\life.inc file defines the CELL data type:

```
type INFRATAG is record
    STRING sKeyName
    INTEGER iIndex
```

For the objects in the UltraWinToolbar, define a new class:

```
winclass ToolBarItem : AnyWin
    // NOTE: no tag
    INFRATAG pseudotag
```

Writing the declaration

Since graphical objects aren't recognized by the SilkTest recording tools, you need to modify the declarations by hand to add the custom objects. You can write a declaration for each of the items in the toolbar as follows:

```
winclass InfraToolBar CustomWin
    tag
    "[Infragistics.Win.UltraWinToolbars.UltraToolbarsDockArea]"
```

```

ToolBarItem GridButton
    INFRATAG pseudotag = {"GridButton",1}
ToolBarItem Cut
    INFRATAG pseudotag = {"Cut",2}

// continue for additional Toolbar items

```

Writing the 4Test prototype

Your extension functions must take as an argument the pseudotag that you have defined. This pseudotag will indicate to your extension function which object the script is referring to. However, the scripter should not need to provide this information while writing scripts. The solution is to write an extension function which requires the pseudotag as an argument, and a 4Test “wrapper” function which supplies it automatically. For example:

```

winclass ToolBarItem : AnyWin
    // NOTE: no tag
    CELL pseudotag

extern void InternalSelect (INFRATAG InfraTag)

void Select ()
    InternalSelect (this.pseudotag)

```

The above declarations and class definitions will allow scripters to write object-oriented and easy-to-read scripts like the following:

```

main ()
    InfragisticsUltraWinToolbars.SetActive()
    // Load the EK DLL into the applications process

    InfragisticsUltraWinToolbars.LoadLibrary("c:\infra_
toolbar.dll")

    InfragisticsUltraWinToolbars.InfragisticsWinUltraWinTool
b1.Copy.Select()

```

Writing the function

The extension function has access to information about the toolbar objects, including:

- The hWnd of the parent window, passed as an argument
- The key name and the index of the toolbar object, passed as an argument (the pseudotag)
- The number of items in the toolbar, from internal data
- The rectangle of each object in the toolbar

With this information, it's possible to calculate the correct pixel coordinates for a mouse click.

```
QAP_RegisterClassFun("ToolBarItem", "InternalSelect",
Select_Item,
                                0, 1, T_LIST_ANYTYPE);

void Select_Item(PARGS pArgs)
{
    PSZ pszKeyName = NULL;
    int iIndex;
    POINT Point;
    LIST Tag;
    System::Drawing::Rectangle rect;
    Control* pControl = NULL;
    HWND hWnd = pArgs->hWnd;

    Tag = GetArg(0, List);

    pszKeyName = GetListItem(Tag,0,pszValue);

    pControl =
System::Windows::Forms::Control::FromHandle(hWnd);
    if (pControl)
    {
        SEGUE_UltraToolBarDockArea* pDockToolBar = new
SEGUE_UltraToolBarDockArea (pControl);
        int iCount = pDockToolBar->get_Count();

        if (!pszKeyName || pszKeyName[0] == '\0')
        {
            iIndex = GetListItem(Tag, 1, lValue) - 1;
            if (iIndex < 0 || iIndex > iCount)
            {
                QAP_RaiseError(1, "Tag index is Invalid");
            }
            rect = pDockToolBar->get_Rect(iIndex);
        }
        else
        {
            rect = pDockToolBar->get_Rect(pszKeyName);
        }

        Point.x = rect.Left;
        Point.y = rect.Top;

        QAP_ClickMouse(hWnd, &Point, 1,EVT_BUTTON1);
    }
    else
```

```

        QAP_RaiseError(1, "Unable to map the window handle
to the .NET control");
    }

```

Automated declarations

As stated in a previous section, SilkTest will not be able to automatically produce declarations for graphical objects. However, you can use the Extension Kit to write a function that will return declarations for your custom objects. These declarations can be pasted into your include files and used in your scripts.

The following code generates a declaration for the toolbar in the Infragistics example:

```

void Create_Decl(PARGS pArgs)
{
    CHAR szWindowName[256];
    CHAR szPseudoTag[256];
    PSZ pszName = NULL;
    Control* pControl = NULL;
    HWND hWnd = pArgs->hWnd;

    pControl =
System::Windows::Forms::Control::FromHandle(hWnd);
    if (pControl)
    {
        SEGUE_UltraToolBarDockArea * pDockToolBar = new
        SEGUE_UltraToolBarDockArea(pControl);
        int iCount = pDockToolBar->get_Count();
        QAP_ReturnListOpen(RETVAl);

        for (int i=0; i < iCount; i++)
        {
            System::String* aString2 = pDockToolBar->get_
Name(i);
            pszName =
(Psz)Marshal::StringToHGlobalAnsi(aString2).ToPointer();
            if (pszName)
            {
                sprintf(szWindowName, "ToolBarItem %s%d",
pszName, i);
                sprintf(szPseudoTag, "\tINFRATAG pseudotag =
{ \"%s\", %d}",
                    pszName, i);
                QAP_ReturnString(RETVAl, szWindowName);
                QAP_ReturnString(RETVAl, szPseudoTag);
            }
        }
    }
}

```

```
        }  
  
        QAP_ReturnListClose (RETVAL);  
    }  
}
```

Internal functionality and data

The Extension Kit allows easy access to all internal functionality and data in your application. Internal functions can be called from 4Test, and values of variables in your application can be returned to scripts. This allows you to test objects with no GUI at all (such as a spreadsheet calculation engine, for example).

Client/server testing

The Extension Kit can also be used to facilitate back-end client/server testing. SilkTest allows you to test the client's GUI, but the Extension Kit allows you to write functions for testing back-end communications for both the client and server.

The Extension Kit provides the tools to write tests against the internal messaging systems of your application. If your server runs on a platform that SilkTest doesn't support, you can use your client application's own internal messaging protocol in order to communicate with and test the server.

Functions that contain one or more arguments

Using a new version of SilkTest causes errors in testcases that call functions created with the Extension Kit. Some functions work, but functions that contain one or more arguments cause the error: **Incorrect number of arguments.**

It is imperative that the version of assist.dll you are using exactly match the version of SilkTest. If you include assist.dll in your application directory, then you must copy in the new version of assist.dll when you change versions of SilkTest.

If you implicitly link assist.lib into your extension DLL, then you must rebuild your extension DLL when you change versions of SilkTest. It will be easier to maintain your extension DLL if you explicitly load assist.dll. For more information, see [“Making the assist.dll accessible” on page 15](#).

3 ADVANCED TOPICS

Functions that contain one or more arguments

4

Function Reference

This chapter contains descriptions of Extension Kit functions. The functions generally perform one of these tasks:

- Register with the SilkTest Agent.
- Register external functions.
- Generate mouse and keyboard events.
- Return values to 4Test.
- Raise 4Test exceptions.

In this chapter

This chapter covers:

Topic	Page
QAP_ClickMouse	48
QAP_Initialize	50
QAP_MoveMouse	51
QAP_PressKeys	52
QAP_PressMouse	54
QAP_RaiseError	55
QAP_RegisterClassFun	57
QAP_RegisterWindowFun	59
QAP_ReleaseKeys	61
QAP_ReleaseMouse	62
QAP_ReturnBoolean	64
QAP_ReturnInteger	65
QAP_ReturnListClose	65

Topic	Page
QAP_ReturnListOpen	66
QAP_ReturnNull	67
QAP_ReturnReal	67
QAP_ReturnString	68
QAP_RouteAllClassFun	70
QAP_RouteAllWindowsFun	71
QAP_Terminate	73
QAP_TypeKeys	74
QAP_UnregisterClassFun	76
QAP_UnregisterWindowFun	78

QAP_ClickMouse

Action Clicks the mouse at the specified coordinates.

Syntax **BOOL** QAP_ClickMouse (**HWND** *hWnd*, **LPPOINT** *pPoint*, **int** *iCount*, **UINT** *uiFlags*)

hWnd The handle of the window on which to operate. This is part of the ARGV structure passed to your function from 4Test.

pPoint A pointer to a structure containing the coordinates of the mouse event. This point is relative to the specified window.

iCount The number of clicks.

uiFlags Flags specifying mouse options, which are listed in the following table:

Flag	Meaning
EVT_BUTTON1	The left mouse button.
EVT_BUTTON2	The right mouse button.
EVT_BUTTON3	The middle mouse button.

Flag	Meaning
EVT_FLUSH	Send each event as it is generated. This forces a flush of each event as it occurs . (By default, mouse events are sent when all pressed buttons have been released.)
EVT_NOFLUSH	Do not send the event immediately, even though all pressed keys have been released. This means that QAP_TypeKeys waits for quiet times before flushing keyboard events. (By default, mouse events are sent when all pressed buttons have been released.)
EVT_NODELAY	Ignore the mouse delay that has been set in SilkTest.

Returns

If the function succeeds, the return value is TRUE; if it fails, the return value is FALSE.

Example

```
void QAPFUNC MC_PerformClick(PARGS pArgs)
{
    POINT Point;

    /* set the coordinates to (20,30) local to the window */
    Point.x = 20;
    Point.y = 30;

    /* click the left button once at those coordinates */
    QAP_ClickMouse (pArgs->hWnd, &Point, 1,
    EVT_BUTTON1);
    /* click the right button 3 times at those coordinates,
    * ignoring the mouse delay set in SilkTest
    */
    QAP_ClickMouse (pArgs->hWnd, &Point, 3,
    EVT_BUTTON2 | EVT_NODELAY)
}
```

Notes

QAP_ClickMouse performs *iCount* clicks in the window specified by *hWnd* at the coordinates specified by *pPoint*. The *uiFlags* argument specifies mouse options and may be a combination of the flags listed in the above table.

See also

[“QAP_MoveMouse” on page 51](#), [“QAP_PressMouse” on page 54](#), [“QAP_ReleaseMouse” on page 62](#)

QAP_Initialize

Action	Registers the extension with the QAP Agent.
Syntax	BOOL QAP_Initialize (VOID)
Returns	If the function succeeds, the return value is TRUE; if it fails, the return value is FALSE.
Example	<pre>long FAR fAgentRunning; fAgentRunning = QAP_Initialize(); if (fAgentRunning) { /* register your functions here */ }</pre>
Notes	QAP_Initialize registers your extension with the QAP Agent. It must be called during your extension's initialization and before any functions are registered.
See also	QAP_Terminate

QAP_MoveMouse

Action Moves the mouse to the specified coordinates.

Syntax **BOOL QAP_MoveMouse (HWND *hWnd*, LPPOINT *pPoint*, UINT *uiFlags*)**

hWnd The handle of the window on which to operate. This is part of the ARGV structure passed to your function from 4Test.

pPoint A pointer to a structure containing the coordinates of the mouse event. This point is relative to the specified window.

uiFlags Flags specifying mouse options, listed in the following table:

Flag	Meaning
EVT_BUTTON1	The left mouse button.
EVT_BUTTON2	The right mouse button.
EVT_BUTTON3	The middle mouse button.
EVT_FLUSH	Send each event as it is generated. This forces a flush of each event as it occurs . (By default, mouse events are sent when all pressed buttons have been released.)
EVT_NOFLUSH	Do not send the event immediately, even though all pressed keys have been released. This means that QAP_TypeKeys waits for quiet times before flushing keyboard events. (By default, mouse events are sent when all pressed buttons have been released.)
EVT_NODELAY	Ignore the mouse delay that has been set in SilkTest.

Returns If the function succeeds, the return value is TRUE; if it fails, the return value is FALSE.

Example

```
void QAPFUNC MC_MouseMove(PARGS pArgs)
{
    POINT Point;

    /* set the coordinates to (20,30) */
    Point.x = 20;
    Point.y = 30;
```

```

    /* move the mouse to those coordinates and send the *
     * event without changing the state of mouse buttons */
    QAP_MoveMouse (pArgs->hWnd, &Point,
        EVT_FLUSH);
}

```

Notes

QAP_MoveMouse moves the mouse to the window specified by *hWnd* at the coordinates specified by *pPoint*. The *uiFlags* argument specifies mouse options and may be a combination of the flags listed in the above table.

QAP_MoveMouse does not press or release any mouse buttons.

By default, mouse events are sent when all pressed buttons have been released. To change from the default Flush behavior, use the EVT_FLUSH or EVT_NOFLUSH flag.

See also

[“QAP_ClickMouse” on page 48](#), [“QAP_MoveMouse” on page 51](#), [“QAP_ReleaseMouse” on page 62](#)

QAP_PressKeys

Action

Presses the specified keys.

Syntax

BOOL QAP_PressKeys (HWND *hWnd*, LPSTR *lpzKeys* LPPOINT *pPoint*, UINT *uiFlags*)

hWnd The handle of the window on which to operate. This is part of the ARGV structure passed to your function from 4Test.

lpzKeys The keys to be pressed.

pPoint A pointer to a structure containing the coordinates of the keyboard event. This point is relative to the specified window.

uiFlags Flags specifying keyboard options, listed in the following table:

Flag	Meaning
EVT_NOPARSING	Type the string verbatim, ignoring special characters.
EVT_FLUSH	Send each event as it is generated. This forces a flush of each event as it occurs . (By default, mouse events are sent when all pressed buttons have been released.)

EVT_NOFLUSH	Do not send the event immediately, even though all pressed keys have been released. This means that QAP_TypeKeys waits for quiet times before flushing keyboard events. (By default, mouse events are sent when all pressed buttons have been released.)
EVT_SETFOCUS	Set the focus to the specified window before sending the events.
EVT_NODELAY	Ignore keyboard delay that has been set in SilkTest.

Returns

If the function succeeds, the return value is TRUE; if it fails, the return value is FALSE.

Example

```
void QAPFUNC MC_PressKeys(PARGS pArgs)
{
    POINT Point;
    /* set the coordinates to (20,30) */
    Point.x = 20;
    Point.y = 30;
    /* press the shift and alt keys at those coordinates, *
     * ignoring the keyboard delay set in SilkTest */
    QAP_PressKeys (pArgs->hWnd,
        "<Shift><Alt>", &Point, EVT_NODELAY);
}
```

Notes

QAP_PressKeys presses the keys *lpzKeys* in the window specified by *hWnd* at the coordinates specified by *pPoint*. The *uiFlags* argument specifies keyboard options and may be a combination of the flags listed in the above table. By default, keyboard events are not sent until all pressed keys have been released. In order to flush keyboard events before all keys have been released, use the EVT_FLUSH flag. For more information about specifying keys by name, see the *4Test Language Reference*.

See also

[“QAP_MoveMouse” on page 51](#), QAP_ReleaseKeys, QAP_TypeKeys

QAP_PressMouse

Action Presses the mouse at the specified coordinates.

Syntax **BOOL QAP_PressMouse (HWND *hWnd*, LPPOINT *pPoint*, UINT *uiFlags*)**

hWnd The handle of the window on which to operate. This is part of the ARGV structure passed to your function from 4Test.

pPoint A pointer to a structure containing the coordinates of the mouse event. This point is relative to the specified window.

uiFlags Flags specifying mouse options, which are listed in the following table:

Flag	Meaning
EVT_BUTTON1	The left mouse button.
EVT_BUTTON2	The right mouse button.
EVT_BUTTON3	The middle mouse button.
EVT_FLUSH	Send each event as it is generated. This forces a flush of each event as it occurs . (By default, mouse events are sent when all pressed buttons have been released.)
EVT_NOFLUSH	Do not send the event immediately, even though all pressed keys have been released. This means that QAP_TypeKeys waits for quiet times before flushing keyboard events. (By default, mouse events are sent when all pressed buttons have been released.)
EVT_NODELAY	Ignore mouse delay that has been set in SilkTest.

Returns If the function succeeds, the return value is TRUE; if it fails, the return value is FALSE.

Example

```
void QAPFUNC MC_MousePress(PARGS pArgs)
{
    POINT Point;

    /* set the coordinates to (20,30) */
    Point.x = 20;
```

```

    Point.y = 30;
    /* press the right button at those coordinates, *
     * ignoring the mouse delay set in */
    QAP_PressMouse (pArgs->hWnd, &Point,
        EVT_BUTTON2 | EVT_NODELAY)
}

```

Notes

QAP_PressMouse presses the mouse in the window specified by *hWnd* at the coordinates specified by *pPoint*. The *uiFlags* argument specifies mouse options and may be a combination of the flags listed in the above table.

By default, mouse events are sent when all pressed buttons have been released. To change from the default Flush behavior, use the EVT_FLUSH or EVT_NOFLUSH flag.

See also

[“QAP_ClickMouse” on page 48](#), [“QAP_MoveMouse” on page 51](#), [“QAP_ReleaseMouse” on page 62](#)

QAP_RaiseError

Action

Raises a 4Test exception with the number and message given.

Four forms of QAP_RaiseError are available, depending upon the type of extension you are using. Note that QAP_RaiseError raises a 4Test exception in the appropriate format, based on the type of extension you are using. For example, if you are using a:

Extension Type...	then QAP_RaiseError raises a 4Test exception in...
Unicode	Unicode format
Multibyte	Multibyte format
ANSI	ANSI format

If you want to raise a 4Test exception in a specific format, you can use the following functions:

- QAP_RaiseErrorA: available for all extensions; raises a 4Test exception in ANSI format
- QAP_RaiseErrorM: available for multibyte extensions; raises a 4Test exception in multibyte format
- QAP_RaiseErrorW: available for unicode extensions; raises a 4Test exception in unicode format. Note that the syntax for QAP_RaiseErrorW is slightly different; see the Syntax section.

These functions are made available by the RaiseError macro. In multibyte and unicode extensions, the RaiseError macro expands to either RaiseErrorM or RaiseErrorW, depending upon the extension type. This occurs automatically, based on the type of extension you are using. Therefore, you can use QAP_RaiseError to raise a 4Test exception automatically in the appropriate format.

Borland recommends that you use QAP_RaiseError, regardless of the type of extension you are using. This function raises a 4Test exception based on the type of extension you are using and makes your extension more flexible and easier to re-use.

Syntax

QAP_RaiseError, **QAP_RaiseErrorA**, and **QAP_RaiseErrorM** all have the same syntax. The following syntax is for QAP_RaiseError. If you are using QAP_RaiseErrorA or QAP_RaiseErrorM, make sure you modify the function name appropriately; the rest of the syntax is the same as following:

VOID QAP_RaiseError (LONG *lError*, LPSTR *lpzFormat*, ...)

Note: The syntax for **QAP_RaiseErrorW** is slightly different; **LPSTR** should be replaced with **LPWSTR** to indicate a wide string. The syntax of QAP_RaiseErrorW is:

VOID QAP_RaiseErrorW (LONG *lError*, LPWSTR *lpzFormat*, ...)

lError The number of the exception.
lpzFormat The format text for the exception, which will be displayed to the user.
 ... The values for the error string.

Returns

This function does not return a value.

Example

```
QAP_RaiseError(1, "Error: my extension found an error!");
long max = 10;
QAP_RaiseError(2, "Error: the maximum value is %d", MAXVAL);
```

Notes

QAP_RaiseError will raise a 4Test exception with a number of *lError* and an error string of *lpzFormat*. The formatting for QAP_RaiseError works like C's printf.

Only one call to QAP_RaiseError should be made within a function. After calling QAP_RaiseError, don't make any calls to the QAP_Return* functions. Doing so will cause an "Agent returned an invalid response" error.

QAP_RegisterClassFun

Action

Registers a class function (method) with the Agent.

Four forms of QAP_RegisterClassFun are available, depending upon the type of extension you are using. Note that QAP_RegisterClassFun registers functions with string parameters in the appropriate format, based on the type of extension you are using. For example, if you are using a:

Extension Type...	then QAP_RegisterClassFun registers the function with string arguments in ...
Unicode	Unicode format
Multibyte	Multibyte format
ANSI	ANSI format

If you want to register a function with string arguments in a specific format, you can use the following functions:

- QAP_RegisterClassFunA: available for all extensions; registers function with ANSI string argument
- QAP_RegisterClassFunM: available for multibyte extensions; registers function with multibyte string argument
- QAP_RegisterClassFunW: available for unicode extensions; registers function with unicode string argument. Note that the syntax for QAP_RegisterClassFunW is slightly different; see the Syntax section.

These functions are made available by the RegisterClassFun macro. In multibyte and unicode extensions, the RegisterClassFun macro expands to either RegisterClassFunM or RegisterClassFunW, depending upon the extension type. This occurs automatically, based on the type of extension you are using. Therefore, you can use QAP_RegisterClassFun to register functions automatically in the appropriate format.

Borland recommends that you use QAP_RegisterClassFun, regardless of the type of extension you are using. This function registers functions with string arguments based on the type of extension you are using and makes your extension more flexible and easier to re-use.

Syntax

QAP_RegisterClassFun, **QAP_RegisterClassFunA**, and **QAP_RegisterClassFunM** all have the same syntax. The following syntax is for **QAP_RegisterClassFun**. If you are using **QAP_RegisterClassFunA** or **QAP_RegisterClassFunM**, make sure you modify the function name appropriately; the rest of the syntax is the same as following:

BOOL QAP_RegisterClassFun (LPSTR *lpzClass*, LPSTR *lpzName*, FQAPFUNC *MyFunc*, TYPE *RetType*, int *iNumParam*, ...)

Note: The syntax for **QAP_RegisterClassFunW** is slightly different; **LPSTR** should be replaced with **LPWSTR** to indicate a wide string. The syntax of **QAP_RegisterClassFunW** is:

BOOL QAP_RegisterClassFunW (LPWSTR *lpzClass*, LPWSTR *lpzName*, FQAPFUNC *MyFunc*, TYPE *RetType*, int *iNumParam*, ...)

lpzClass The name of the class.
lpzName The name of the 4Test method.
MyFunc The pointer to the function within the extension.
RetType The 4Test type of the 4Test return value.
iNumParam The number of parameters to the 4Test method.
... The parameter attributes and 4Test type for each argument.

Returns

If the function succeeds, the return value is TRUE; if it fails, the return value is FALSE.

Example

The following example registers a 4Test class function called “MyFunction” for the class “MyClass.” The extension’s internal name for the function is “MC_MyFunction1,” the function returns an integer, and takes no arguments.

```
QAP_RegisterClassFun (_T("MyClass"), _T("MyFunction1"), MC_MyFunction1, T_INTEGER, 0)
```

The following example registers a function that returns VOID and takes two arguments: the first argument is of the 4Test type “in integer” and the second is of the 4Test type “out string”.

```
QAP_RegisterClassFun (_T("MyClass"), _T("MyFunction2"), MC_MyFunction2, T_VOID, 2, P_IN|T_INTEGER, P_OUT|T_STRING);
```

Notes

QAP_RegisterClassFun registers a function for class *lpzClass* with a 4Test method name of *lpzName*, an internal name of *MyFunc*, a return type of *RetType*, and *iNumParam* parameters. For each parameter, you must specify parameter attributes and 4Test type information. This is the equivalent of adding a 4Test function inside a window class definition, except that your function is implemented in C.

For information about 4Test types and parameter attributes, see “[4Test data types](#)” on page 21 and “[Parameter attributes](#)” on page 22.

See also

“[QAP_RegisterWindowFun](#)” on page 59, “[QAP_RouteAllClassFun](#)” on page 70

QAP_RegisterWindowFun

Action

Registers a window (instance) function (method) with the Agent.

Four forms of QAP_RegisterWindowFun are available, depending upon the type of extension you are using. Note that QAP_RegisterWindowFun registers window functions with string parameters in the appropriate format, based on the type of extension you are using. For example, if you are using a:

Extension Type...	then QAP_RegisterWindowFun registers the window function with string arguments in ...
Unicode	Unicode format
Multibyte	Multibyte format
ANSI	ANSI format

If you want to register a window function with string arguments in a specific format, you can use the following functions:

- QAP_RegisterWindowFunA: available for all extensions; registers window function with ANSI string argument
- QAP_RegisterWindowFunM: available for multibyte extensions; registers window function with multibyte string argument
- QAP_RegisterWindowFunW: available for unicode extensions; registers window function with unicode string argument. Note that the syntax for QAP_RegisterWindowFunW is slightly different; see the Syntax section.

These functions are made available by the RegisterWindowFun macro. In multibyte and unicode extensions, the RegisterWindowFun macro expands to either RegisterWindowFunM or RegisterWindowFunW, depending upon the extension type. This occurs automatically, based on the type of extension you are using. Therefore, you can use QAP_RegisterWindowFun to register window functions automatically in the appropriate format.

Borland recommends that you use QAP_RegisterWindowFun, regardless of the type of extension you are using. This function registers functions with string arguments based on the type of extension you are using and makes your extension more flexible and easier to re-use.

Syntax

QAP_RegisterWindowFun, **QAP_RegisterWindowFunA**, and **QAP_RegisterWindowFunM** all have the same syntax. The following syntax is for **QAP_RegisterWindowFun**. If you are using **QAP_RegisterWindowFunA** or **QAP_RegisterWindowFunM**, make sure you modify the function name appropriately; the rest of the syntax is the same as following:

BOOL QAP_RegisterWindowFun (LPSTR *lpzWindow*, LPSTR *lpzName*, FQAPFUNC *MyFunc*, TYPE *RetType*, int *iNumParam*, ...)

Note: The syntax for **QAP_RegisterWindowFunW** is slightly different; **LPSTR** should be replaced with **LPWSTR** to indicate a wide string. The syntax of **QAP_RegisterWindowFunW** is:

BOOL QAP_RegisterWindowFunW (LPWSTR *lpzClass*, LPWSTR *lpzName*, FQAPFUNC *MyFunc*, TYPE *RetType*, int *iNumParam*, ...)

lpzWindow The window identifier.

lpzName The name of the 4Test method.

MyFunc The pointer to the function within the extension.

RetType The 4Test type of the 4Test return value.

iNumParam The number of parameters to the 4Test method.

... The parameter attributes and 4Test type for each argument.

Returns

If the function succeeds, the return value is TRUE; if it fails, the return value is FALSE.

Example

This example registers a 4Test window function called **MyFunction** for the window **MyWindow**. The extension's internal name for the function is **MC_MyFunction1**, the function returns an integer, and takes no arguments.

```
QAP_RegisterWindowFun (_T("MyWindow"), _T("MyFunction1"),  
MC_MyFunction1, T_INTEGER, 0)
```

The following example registers a function that returns **VOID** and takes two arguments: the first argument is of the 4Test type "in integer" and the second is of the 4Test type "out string".

```
QAP_RegisterWindowFun (_T("MyWindow"), _T("MyFunction2"),  
MC_MyFunction2, T_VOID, 2, P_IN|T_INTEGER, P_OUT|T_STRING));
```

Notes

QAP_RegisterWindowFun registers a function for class *lpzWindow* with a 4Test method name of *lpzName*, an internal name of *MyFunc*, a return type of *RetType*, and *iNumParam* parameters. For each parameter, you must specify parameter attributes and 4Test type information. This is the equivalent of adding a 4Test function inside a window declaration, except that your function is implemented in C. For information about 4Test types and parameter attributes, see ["4Test data types" on page 21](#) and ["Parameter attributes" on page 22](#).

See also [“QAP_RouteAllWindowsFun” on page 71](#), [“QAP_UnregisterWindowFun” on page 78](#)

QAP_ReleaseKeys

Action Releases the specified keys.

Syntax **BOOL QAP_ReleaseKeys (HWND *hWnd*, LPSTR *lpszKeys*, LPPOINT *pPoint*, UINT *uiFlags*)**
hWnd The handle of the window on which to operate. This is part of the ARGV structure passed to your function from 4Test.
lpszKeys The keys to be released.
pPoint A pointer to a structure containing the coordinates of the keyboard event or NULL. This point is relative to the specified window.
uiFlags Flags specifying keyboard options, which are listed in the following table:

Flag	Meaning
EVT_NOPARSING	Type the string verbatim, ignoring special characters.
EVT_FLUSH	Send each event as it is generated. This forces a flush of each event as it occurs . (By default, mouse events are sent when all pressed buttons have been released.)
EVT_NOFLUSH	Do not send the event immediately, even though all pressed keys have been released. This means that QAP_TypeKeys waits for quiet times before flushing keyboard events. (By default, mouse events are sent when all pressed buttons have been released.)
EVT_SETFOCUS	Set the focus to the specified window before sending the events.
EVT_NODELAY	Ignore keyboard delay that has been set in SilkTest.

Returns If the function succeeds, the return value is TRUE; if it fails, the return value is FALSE.

Example

```

void QAPFUNC MC_PressAndReleaseKeys(PARGS pArgs)
{
    POINT Point;

    /* set the coordinates to (20,30) */
    Point.x = 20;
    Point.y = 30;

    /* press the shift and alt keys at those coordinates, *
     * ignoring the mouse delay set in */
    QAP_PressKeys (pArgs->hWnd,
        "<Shift><Alt>", &Point, EVT_NODELAY);

    /* release those keys without sending the event */
    QAP_PressKeys (pArgs->hWnd,
        "<Shift><Alt>", &Point, EVT_NOFLUSH);
}

```

Notes

QAP_ReleaseKeys releases the keys *lpszKeys* in the window specified by *hWnd* at the coordinates specified by *pPoint*. The *uiFlags* argument specifies keyboard options and may be a combination of the flags listed in the above table. By default, keyboard events are sent when all pressed keys have been released. In order to queue keyboard events even when all keys have been released, use the EVT_NOFLUSH flag. For more information about specifying keys by name, see the *4Test Language Reference*.

See also

[“QAP_PressKeys” on page 52](#), [“QAP_TypeKeys” on page 74](#)

QAP_ReleaseMouse

Action

Moves the mouse to the specified coordinates and releases it.

Syntax

BOOL QAP_ReleaseMouse (HWND *hWnd*, LPPOINT *pPoint*, UINT *uiFlags*)

hWnd The handle of the window on which to operate. This is part of the ARGV structure passed to your function from 4Test.

pPoint A pointer to a structure containing the coordinates of the mouse event. This point is relative to the specified window.

uiFlags Flags specifying mouse options, which are listed in the following table:

Flag	Meaning
EVT_BUTTON1	The left mouse button.

Flag	Meaning
EVT_BUTTON2	The right mouse button.
EVT_BUTTON3	The middle mouse button.
EVT_FLUSH	Send each event as it is generated. This forces a flush of each event as it occurs . (By default, mouse events are sent when all pressed buttons have been released.)
EVT_NOFLUSH	Do not send the event immediately, even though all pressed keys have been released. This means that QAP_TypeKeys waits for quiet times before flushing keyboard events. (By default, mouse events are sent when all pressed buttons have been released.)
EVT_NODELAY	Ignore mouse delay that has been set in SilkTest.

Returns

If the function succeeds, the return value is TRUE; if it fails, the return value is FALSE.

Example

```
void QAPFUNC MC_MousePressAndRelease(PARGS pArgs)
{
    POINT Point;

    /* set the coordinates to (20,30) */
    Point.x = 20;
    Point.y = 30;

    /* press the right button at those coordinates */
    QAP_PressMouse (pArgs->hWnd, &Point,
    EVT_BUTTON2)

    /* set the coordinates to (40,20) */
    Point.x = 20;
    Point.y = 30;

    /* release the right button at the new coordinates, *
    * ignoring the mouse delay set in */
    QAP_PressMouse (pArgs->hWnd, &Point,
    EVT_BUTTON2 | EVT_NODELAY)
}
```

Notes

QAP_ReleaseMouse moves the mouse to the window specified by *hWnd* at the coordinates specified by *pPoint* and releases the specified button(s). The *uiFlags* argument specifies mouse options and may be a combination of the flags listed in the above table.

By default, mouse events are sent when all pressed buttons have been released. To change from the default Flush behavior, use the EVT_FLUSH or EVT_NOFLUSH flag.

See also

[“QAP_ClickMouse” on page 48](#), [“QAP_MoveMouse” on page 51](#), [“QAP_PressMouse” on page 54](#)

QAP_ReturnBoolean**Action**

Returns a Boolean value to 4Test.

Syntax

VOID QAP_ReturnBoolean (int *iArg*, BOOL *fValue*)

iArg The number of the argument in which to return the value or RETVAL.

fValue The return value.

Returns

This function does not return a value.

Example

```
/* return FALSE as the return value of the 4Test function */
QAP_ReturnBoolean(RETVAL, FALSE);
```

```
/* set the "out" argument in argument 2 to TRUE */
QAP_ReturnBoolean(2, TRUE);
```

Notes

QAP_ReturnBoolean returns the value *fValue* to 4Test. If the argument number of an “out” argument is specified in *iArg*, QAP_ReturnBoolean will set the value of that argument. If *iArg* is RETVAL, QAP_ReturnBoolean will set the value of the 4Test function’s return value. To return a Boolean value inside of a record or list, call QAP_ReturnListOpen before calling this function.

See also

[“QAP_ReturnListClose” on page 65](#), [“QAP_ReturnListOpen” on page 66](#)

QAP_ReturnInteger

Action	Returns an integer value to 4Test.
Syntax	VOID QAP_ReturnInteger (int <i>iArg</i>, LONG <i>IValue</i>) <i>iArg</i> The number of the argument in which to return the value or RETVAL. <i>IValue</i> The integer return value.
Returns	This function does not return a value.
Example	<pre>/* return 17 as the return value of the 4Test function */ QAP_ReturnInteger (RETVAL, 17); /* set the "out" argument in argument 2 to 22 */ QAP_ReturnInteger (2, 22);</pre>
Notes	QAP_ReturnInteger returns the long integer value <i>IValue</i> to 4Test. 4Test uses 32-bit signed integers, hence the LONG <i>IValue</i> . If the argument number of an "out" argument is specified in <i>iArg</i> , QAP_ReturnInteger will set the value of that argument. If <i>iArg</i> is RETVAL, QAP_ReturnInteger will set the value of the 4Test function's return value. To return an integer value inside of a record or list, call QAP_ReturnListOpen before calling this function.
See also	"QAP_ReturnListClose" on page 65 , "QAP_ReturnListOpen" on page 66

QAP_ReturnListClose

Action	Returns the end of a list or record to 4Test.
Syntax	VOID QAP_ReturnListClose (int <i>iArg</i>) <i>iArg</i> The number of the argument in which to return the list (or record) or RETVAL.
Returns	This function does not return a value.
Example	<pre>/* This returns a list of {"a", "b", "c"} */ QAP_ReturnListOpen (RETVAL) QAP_ReturnString (RETVAL, "a"); QAP_ReturnString (RETVAL, "b"); QAP_ReturnString (RETVAL, "c"); QAP_ReturnListClose (RETVAL)</pre>

Notes

QAP_ReturnListClose returns the end of a list or record to 4Test. The elements of the list or record are returned with calls to the other QAP_Return* functions. The beginning of the list is returned with a call to QAP_ReturnListOpen. If the argument number of an **out** argument is specified in *iArg*, QAP_ReturnListClose will set the value of that argument. If *iArg* is RETVAL, QAP_ReturnListClose will set the value of the 4Test function's return value. This function may be nested for returning a list of list value.

See also

[“QAP_ReturnListOpen” on page 66](#)

QAP_ReturnListOpen

Action

Returns the beginning of a list or record to 4Test.

Syntax

VOID QAP_ReturnListOpen (int *iArg*)

iArg The number of the argument in which to return the list (or record) or RETVAL.

Returns

This function does not return a value.

Example

```
/* This returns a list of {"a", "b", "c"} */
QAP_ReturnListOpen(RETVAL)
    QAP_ReturnString(RETVAL, "a");
    QAP_ReturnString(RETVAL, "b");
    QAP_ReturnString(RETVAL, "c");
QAP_ReturnListClose(RETVAL)
```

Notes

QAP_ReturnListOpen returns the beginning of a list or record to 4Test. The elements of the list or record are returned with calls to the other QAP_Return* functions. The end of the list is returned with a call to QAP_ReturnListClose. If the argument number of an **out** argument is specified in *iArg*, QAP_ReturnListOpen will set the value of that argument. If *iArg* is RETVAL, QAP_ReturnListOpen will set the value of the 4Test function's return value. This function may be nested for returning a list of list value.

See also

[“QAP_ReturnListClose” on page 65](#)

QAP_ReturnNull

Action	Returns a value to 4Test.
Syntax	VOID QAP_ReturnNull (int <i>iArg</i>) <i>iArg</i> The number of the argument in which to return the value or RETVAL.
Returns	This function does not return a value.
Example	<pre>/* return NULL as the return value of the 4Test function */ QAP_ReturnNull (RETVAL); /* set the "out" argument in argument 2 to NULL */ QAP_ReturnNull (2);</pre>
Notes	QAP_ReturnNull returns a value to 4Test. If the argument number of an out argument is specified in <i>iArg</i> , QAP_ReturnNull will set the value of that argument. If <i>iArg</i> is RETVAL, QAP_ReturnNull will set the value of the 4Test function's return value. To return a null value inside of a record or list, call QAP_ReturnListOpen before calling this function.
See also	"QAP_ReturnListClose" on page 65 , "QAP_ReturnListOpen" on page 66

QAP_ReturnReal

Action	Returns a real value to 4Test.
Syntax	VOID QAP_ReturnReal (int <i>iArg</i>, DOUBLE <i>dblValue</i>) <i>iArg</i> The number of the argument in which to return the value or RETVAL. <i>dblValue</i> The double return value.
Returns	This function does not return a value.
Example	<pre>/* return 4.3 as the return value of the 4Test function */ QAP_ReturnReal (RETVAL, 4.3); /* set the "out" argument in argument 2 to 7.34 */ QAP_ReturnReal (2, 7.34);</pre>

Notes

QAP_ReturnReal returns the real value *dblValue* to 4Test. If the argument number of an **out** argument is specified in *iArg*, QAP_ReturnReal will set the value of that argument. If *iArg* is RETVAL, QAP_ReturnReal will set the value of the 4Test function's return value. To return a real value inside of a record or list, call QAP_ReturnListOpen before calling this function.

See also

[“QAP_ReturnListClose” on page 65](#), [“QAP_ReturnListOpen” on page 66](#)

QAP_ReturnString

Action

Returns a string or enumerated value to 4Test.

Four forms of QAP_ReturnString are available, depending upon the type of extension you are using. Note that QAP_ReturnString returns a string or enumerated value in the appropriate format, based on the type of extension you are using. For example, if you are using a:

Extension Type...	then QAP_ReturnString returns a string or enumerated value in...
Unicode	Unicode format
Multibyte	Multibyte format
ANSI	ANSI format

If you want to return a string or enumerated value in a specific format, you can use the following functions:

- QAP_ReturnStringA: available for all extensions; returns a string or enumerated value in ANSI format
- QAP_ReturnStringM: available for multibyte extensions; returns a string or enumerated value in multibyte format
- QAP_ReturnStringW: available for unicode extensions; returns a string or enumerated value in unicode format. Note that the syntax for QAP_ReturnStringW is slightly different; see the Syntax section.

These functions are made available by the ReturnString macro. In multibyte and unicode extensions, the ReturnString macro expands to either ReturnStringM or ReturnStringW, depending upon the extension type. This occurs automatically, based on the type of extension you are using. Therefore, you can use QAP_ReturnString to return a string or enumerated value automatically in the appropriate format.

Borland recommends that you use QAP_ReturnString, regardless of the type of extension you are using. This function returns strings or enumerated values based on the type of extension you are using and makes your extension more flexible and easier to re-use.

Syntax

QAP_ReturnString, **QAP_ReturnStringA**, and **QAP_ReturnStringM** all have the same syntax. The following syntax is for **QAP_ReturnString**. If you are using **QAP_ReturnStringA** or **QAP_ReturnStringM**, make sure you modify the function name appropriately; the rest of the syntax is the same as following:

VOID QAP_ReturnString (int *iArg*, LPSTR *lpszValue*)

Note: The syntax for **QAP_ReturnStringW** is slightly different; **LPSTR** should be replaced with **LPWSTR** to indicate a wide string. The syntax of **QAP_ReturnStringW** is:

VOID QAP_ReturnStringW (int *iArg*, LPWSTR *lpszValue*)

iArg The number of the argument in which to return the value or RETVAL.

lpszValue The string return value.

Returns

This function does not return a value.

Example

```
/* return "hello" as the return value of the 4Test
function*/
QAP_ReturnString(RETVAL, "hello");

/* set the "out" argument in argument 2 to "goodbye" */
QAP_ReturnString(2, "goodbye");
```

Notes

QAP_ReturnString returns the string value *lpszValue* to **4Test**. If the argument number of an **out** argument is specified in *iArg*, **QAP_ReturnString** will set the value of that argument. If *iArg* is **RETVAL**, **QAP_ReturnString** will set the value of the **4Test** function's return value. To return a string value inside of a record or list, call **QAP_ReturnListOpen** before calling this function.

See also

[“QAP_ReturnListClose” on page 65](#), [“QAP_ReturnListOpen” on page 66](#)

QAP_RouteAllClassFun

Action

Tells the Agent to route all external functions for the specified class to the extension.

Four forms of QAP_RouteAllClassFun are available, depending upon the type of extension you are using. Note that QAP_RouteAllClassFun routes external functions for the specified class in the appropriate format, based on the type of extension you are using. For example, if you are using a:

Extension Type...	then QAP_RouteAllClassFun routes external functions for the specified class in ...
Unicode	Unicode format
Multibyte	Multibyte format
ANSI	ANSI format

If you want to route all external functions for the specified class in a specific format, you can use the following functions:

- QAP_RouteAllClassFunA: available for all extensions; routes all external functions for the specified class in ANSI format
- QAP_RouteAllClassFunM: available for multibyte extensions; routes all external functions for the specified class in multibyte format
- QAP_RouteAllClassFunW: available for unicode extensions; routes all external functions for the specified class in unicode format. Note that the syntax for QAP_RouteAllClassFunW is slightly different; see the Syntax section.

These functions are made available by the RouteAllClassFun macro. In multibyte and unicode extensions, the RouteAllClassFun macro expands to either RouteAllClassFunM or RouteAllClassFunW, depending upon the extension type. This occurs automatically, based on the type of extension you are using. Therefore, you can use QAP_RouteAllClassFun to route external functions for the specified class in the appropriate format.

Borland recommends that you use QAP_RouteAllClassFun, regardless of the type of extension you are using. This function routes external functions for the specified class in the appropriate format based on the type of extension you are using and makes your extension more flexible and easier to re-use.

Syntax	<p>QAP_RouteAllClassFun, QAP_RouteAllClassFunA, and QAP_RouteAllClassFunM all have the same syntax. The following syntax is for QAP_RouteAllClassFun. If you are using QAP_RouteAllClassFunA or QAP_RouteAllClassFunM, make sure you modify the function name appropriately; the rest of the syntax is the same as following:</p> <p>BOOL QAP_RouteAllClassFun (LPSTR <i>lpzClass</i>)</p> <p>Note: The syntax for QAP_RouteAllClassFunW is slightly different; LPSTR should be replaced with LPWSTR to indicate a wide string. The syntax of QAP_RouteAllClassFunW is:</p> <p>BOOL QAP_RouteAllClassFunW (LPWSTR <i>lpzClass</i>)</p> <p><i>lpzClass</i> The name of the 4Test class.</p>
Returns	If the function succeeds, the return value is TRUE; if it fails, the return value is FALSE.
Example	<pre>QAP_RouteAllClassFun (_T("MyClass"));</pre>
Notes	QAP_RouteAllClassFun() tells the Agent to route all external functions for the class <i>lpzClass</i> to the extension. All functions for the class <i>lpzClass</i> must be registered individually using QAP_RegisterClassFun() .
See also	“QAP_RegisterClassFun” on page 57 , “QAP_RouteAllWindowsFun” on page 71

QAP_RouteAllWindowsFun

Action	<p>Tells the Agent to route all external functions for the specified window to the extension.</p> <p>Four forms of QAP_RouteAllWindowsFun are available, depending upon the type of extension you are using. Note that QAP_RouteAllWindowsFun routes external functions for the specified window in the appropriate format, based on the type of extension you are using. For example, if you are using a:</p> <table> <tr> <td style="padding-right: 20px;">Extension Type...</td> <td>then QAP_RouteAllWindowsFun routes external functions for the window in ...</td> </tr> <tr> <td>Unicode</td> <td>Unicode format</td> </tr> <tr> <td>Multibyte</td> <td>Multibyte format</td> </tr> <tr> <td>ANSI</td> <td>ANSI format</td> </tr> </table> <p>If you want to route all external functions for the specified window in a specific format, you can use the following functions:</p> <ul style="list-style-type: none"> • QAP_RouteAllWindowsFunA: available for all extensions; routes all external functions for the specified window in ANSI format 	Extension Type...	then QAP_RouteAllWindowsFun routes external functions for the window in ...	Unicode	Unicode format	Multibyte	Multibyte format	ANSI	ANSI format
Extension Type...	then QAP_RouteAllWindowsFun routes external functions for the window in ...								
Unicode	Unicode format								
Multibyte	Multibyte format								
ANSI	ANSI format								

- QAP_RouteAllWindowsFunM: available for multibyte extensions; routes all external functions for the specified window in multibyte format
- QAP_RouteAllWindowsFunW: available for unicode extensions; routes all external functions for the specified window in unicode format. Note that the syntax for QAP_RouteAllWindowsFunW is slightly different; see the Syntax section.

These functions are made available by the RouteAllWindowsFun macro. In multibyte and unicode extensions, the RouteAllWindowsFun macro expands to either RouteAllWindowsFunM or RouteAllWindowsFunW, depending upon the extension type. This occurs automatically, based on the type of extension you are using. Therefore, you can use QAP_RouteAllWindowsFun to route external functions for the specified window in the appropriate format.

Borland recommends that you use QAP_RouteAllWindowsFun, regardless of the type of extension you are using. This function routes external functions for the specified window in the appropriate format based on the type of extension you are using and makes your extension more flexible and easier to re-use.

Syntax

QAP_RouteAllWindowsFun, **QAP_RouteAllWindowsFunA**, and **QAP_RouteAllWindowsFunM** all have the same syntax. The following syntax is for QAP_RouteAllWindowsFun. If you are using QAP_RouteAllWindowsFunA or QAP_RouteAllWindowsFunM, make sure you modify the function name appropriately; the rest of the syntax is the same as following:

BOOL QAP_RouteAllWindowsFun (LPSTR *lpzWindow*)

Note: The syntax for **QAP_RouteAllWindowsFunW** is slightly different; **LPSTR** should be replaced with **LPWSTR** to indicate a wide string. The syntax of QAP_RouteAllWindowsFunW is:

BOOL QAP_RouteAllWindowsFunW (LPWSTR *lpzWindow*)

lpzWindow The window identifier.

Returns

If the function succeeds, the return value is TRUE; if it fails, the return value is FALSE.

Example

```
QAP_RouteAllWindowsFun (_T("MyWindow"));
```

Notes

QAP_RouteAllWindowsFun tells the Agent to route all external functions for the class *lpzWindow* to the extension. All functions for the class *lpzWindow* must be registered individually using QAP_RegisterWindowFun.

See also

[“QAP_RegisterWindowFun” on page 59](#), [“QAP_RouteAllWindowsFun” on page 70](#)

QAP_Terminate

Action	Unregisters the extension with the Agent.
Syntax	BOOL QAP_Terminate (VOID)
Returns	If the function succeeds, the return value is TRUE; if it fails, the return value is FALSE.
Example	<pre>/* initialization */ BOOL FAR fAgentRunning; fAgentRunning = QAP_Initialize(); /* termination */ if (fAgentRunning) { /* register your functions here */ }</pre>
Notes	QAP_Terminate unregisters your extension with the Agent and should be called during your extension's termination. QAP_Terminate will unregister any functions you have registered with calls to QAP_RegisterClassFun or QAP_RegisterWindowFun.
See also	“QAP_Initialize” on page 50

QAP_TypeKeys

Action Sends the specified keystrokes.

Four forms of QAP_TypeKeys are available, depending upon the type of extension you are using. Note that QAP_TypeKeys sends the specified keystrokes in the appropriate format, based on the type of extension you are using. For example, if you are using a:

Extension Type...	then QAP_TypeKeys sends the specified keystrokes in ...
Unicode	Unicode format
Multibyte	Multibyte format
ANSI	ANSI format

If you want to send the specified keystrokes in a specific format, you can use the following functions:

- QAP_TypeKeysA: available for all extensions; sends the specified keystrokes in ANSI format
- QAP_TypeKeysM: available for multibyte extensions; sends the specified keystrokes in multibyte format
- QAP_TypeKeysW: available for unicode extensions; sends the specified keystrokes in unicode format. Note that the syntax for QAP_TypeKeysW is slightly different; see the Syntax section.

These functions are made available by the TypeKeys macro. In multibyte and unicode extensions, the TypeKeys macro expands to either TypeKeysM or TypeKeysW, depending upon the extension type. This occurs automatically, based on the type of extension you are using. Therefore, you can use QAP_TypeKeys to send the specified keystrokes in the appropriate format.

Borland recommends that you use QAP_TypeKeys, regardless of the type of extension you are using. This function sends the specified keystrokes in the appropriate format based on the type of extension you are using and makes your extension more flexible and easier to re-use.

Syntax

QAP_TypeKeys, **QAP_TypeKeysA**, and **QAP_TypeKeysM** all have the same syntax. The following syntax is for **QAP_TypeKeys**. If you are using **QAP_TypeKeysA** or **QAP_TypeKeysM**, make sure you modify the function name appropriately; the rest of the syntax is the same as following:

BOOL QAP_TypeKeys (HWND hWnd, LPSTR lpszKeysLPPOINT pPoint, UINT uiFlags)

Note: The syntax for **QAP_TypeKeysW** is slightly different; **LPSTR** should be replaced with **LPWSTR** to indicate a wide string. The syntax of **QAP_TypeKeysW** is:

BOOL QAP_TypeKeysW (HWND hWnd, LPWSTR lpszKeysLPPOINT pPoint, UINT uiFlags)

hWnd The handle of the window on which to operate. This is part of the ARGV structure passed to your function from 4Test.

lpszKeys The keys to be pressed.

pPoint A pointer to a structure containing the coordinates of the keyboard event. This point is relative to the specified window.

uiFlags Flags specifying keyboard options, which are listed in the following table

Flag	Meaning
EVT_NOPARSING	Type the string verbatim, ignoring special characters.
EVT_FLUSH	Send each event as it is generated. This forces a flush of each event as it occurs . (By default, mouse events are sent when all pressed buttons have been released.)
EVT_NOFLUSH	Do not send the event immediately, even though all pressed keys have been released. This means that QAP_TypeKeys waits for quiet times before flushing keyboard events. (By default, mouse events are sent when all pressed buttons have been released.)
EVT_SETFOCUS	Set the focus to the specified window before sending the events.
EVT_NODELAY	Ignore keyboard delay that has been set in SilkTest.

Returns If the function succeeds, the return value is TRUE; if it fails, the return value is FALSE.

Example

```
void QAPFUNC MC_DoTyping(PARGS pArgs)
{
    POINT Point;
    /* set the coordinates to (20,30) */
    Point.x = 20;
    Point.y = 30;
    /* type "hello" at those coordinates,          *
     * ignoring the mouse delay set in */
    QAP_TypeKeys (pArgs->hWnd, "hello",
        &Point, EVT_NODELAY);
}
```

Notes QAP_TypeKeys types *lpzKeys* in the window specified by *hWnd* at the coordinates specified by *pPoint*. The *uiFlags* argument specifies keyboard options and may be a combination of the flags listed in the above table. By default, keyboard events are sent when all pressed keys have been released. In order to queue keyboard events even when all buttons have been released, use the EVT_NOFLUSH flag. For more information about specifying keys by name, see the *4Test Language Reference*.

See also [“QAP_PressKeys” on page 52](#), [“QAP_ReleaseKeys” on page 61](#)

QAP_UnregisterClassFun

Action Unregisters a class function with the Agent.

Four forms of QAP_UnregisterClassFun are available, depending upon the type of extension you are using. Note that QAP_UnregisterClassFun unregisters functions with string parameters in the appropriate format, based on the type of extension you are using. For example, if you are using a:

Extension Type...	then QAP_UnregisterClassFun unregisters the function with string arguments in ...
Unicode	Unicode format
Multibyte	Multibyte format
ANSI	ANSI format

If you want to unregister a function with string arguments in a specific format, you can use the following functions:

- QAP_UnregisterClassFunA: available for all extensions; unregisters function with ANSI string argument

- QAP_UnregisterClassFunM: available for multibyte extensions; unregisters function with multibyte string argument
- QAP_UnregisterClassFunW: available for unicode extensions; unregisters function with unicode string argument. Note that the syntax for QAP_UnregisterClassFunW is slightly different; see the Syntax section.

These functions are made available by the UnregisterClassFun macro. In multibyte and unicode extensions, the UnregisterClassFun macro expands to either UnregisterClassFunM or UnregisterClassFunW, depending upon the extension type. This occurs automatically, based on the type of extension you are using. Therefore, you can use QAP_UnregisterClassFun to unregister functions automatically in the appropriate format.

Borland recommends that you use QAP_UnregisterClassFun, regardless of the type of extension you are using. This function unregisters functions with string arguments based on the type of extension you are using and makes your extension more flexible and easier to re-use.

Syntax

QAP_UnregisterClassFun, **QAP_UnregisterClassFunA**, and **QAP_UnregisterClassFunM** all have the same syntax. The following syntax is for QAP_UnregisterClassFun. If you are using QAP_UnregisterClassFunA or QAP_UnregisterClassFunM, make sure you modify the function name appropriately; the rest of the syntax is the same as following:

BOOL QAP_UnregisterClassFun (LPSTR *lpzClass*, LPSTR *lpzName*)

Note: The syntax for **QAP_UnregisterClassFunW** is slightly different; **LPSTR** should be replaced with **LPWSTR** to indicate a wide string. The syntax of QAP_RegisterClassFunW is:

BOOL QAP_UnregisterClassFun (LPWSTR *lpzClass*, LPWSTR *lpzName*)

*lpzClass*The name of the 4Test class.

*lpzName*The name of the 4Test class method.

Returns

If the function succeeds, the return value is TRUE; if it fails, the return value is FALSE.

Example

```
QAP_UnregisterClassFun (_T("MyClass"), _T("MyFunction"))
```

Notes

QAP_UnregisterClassFun unregisters the function *lpzName* in the class *lpzClass* from the Agent. QAP_Terminate automatically unregisters all functions.

See also [“QAP_RegisterClassFun” on page 57](#), [“QAP_Terminate” on page 73](#),
[“QAP_UnregisterWindowFun” on page 78](#)

QAP_UnregisterWindowFun

Action Unregisters a window function with the Agent.

Four forms of QAP_UnregisterWindowFun are available, depending upon the type of extension you are using. Note that QAP_UnregisterWindowFun unregisters window functions with string parameters in the appropriate format, based on the type of extension you are using. For example, if you are using a:

	then QAP_UnregisterWindowFun unregisters the window function with string arguments in ...
Extension Type...	
Unicode	Unicode format
Multibyte	Multibyte format
ANSI	ANSI format

If you want to unregister a window function with string arguments in a specific format, you can use the following functions:

- QAP_UnregisterWindowFunA: available for all extensions; unregisters window function with ANSI string argument
- QAP_UnregisterWindowFunM: available for multibyte extensions; unregisters window function with multibyte string argument
- QAP_UnregisterWindowFunW: available for unicode extensions; unregisters window function with unicode string argument. Note that the syntax for QAP_UnregisterWindowFunW is slightly different; see the Syntax section.

These functions are made available by the UnregisterWindowFun macro. In multibyte and unicode extensions, the UnregisterWindowFun macro expands to either UnregisterWindowFunM or UnregisterWindowFunW, depending upon the extension type. This occurs automatically, based on the type of extension you are using. Therefore, you can use QAP_UnregisterWindowFun to unregister window functions automatically in the appropriate format.

Borland recommends that you use QAP_UnregisterWindowFun, regardless of the type of extension you are using. This function unregisters window functions with string arguments based on the type of extension you are using and makes your extension more flexible and easier to re-use.

Syntax **QAP_UnregisterWindowFun**, **QAP_UnregisterWindowFunA**, and **QAP_UnregisterWindowFunM** all have the same syntax. The following syntax is for **QAP_UnregisterWindowFun**. If you are using **QAP_UnregisterWindowFunA** or **QAP_UnregisterWindowFunM**, make sure you modify the function name appropriately; the rest of the syntax is the same as following:

BOOL QAP_UnregisterWindowFun (LPSTR *lpzWindow*, LPSTR *lpzName*)

Note: The syntax for **QAP_UnregisterWindowFunW** is slightly different; **LPSTR** should be replaced with **LPWSTR** to indicate a wide string. The syntax of **QAP_UnregisterWindowFunW** is:

BOOL QAP_UnregisterWindowFunW (LPWSTR *lpzWindow*, LPWSTR *lpzName*)

lpzWindow The window identifier.

lpzName The name of the 4Test window method.

Returns If the function succeeds, the return value is TRUE; if it fails, the return value is FALSE.

Example

```
QAP_UnregisterWindowFun (_T("MyWindow"), _T("MyFunction"));
```

Notes **QAP_UnregisterWindowFun** unregisters the function *lpzName* in the window *lpzWindow* from the Agent. **QAP_Terminate** automatically unregisters all functions.

See also [“QAP_RegisterWindowFun” on page 59](#), [“QAP_Terminate” on page 73](#), [“QAP_UnregisterClassFun” on page 76](#)

5

Macro Reference

This chapter describes the macros you use to simplify the retrieval of argument types and values from the ARGV structure that is passed to each extension function.

In this chapter

This chapter covers:

Topic	Page
GetArg	82
GetArgOpt	83
GetArgType	84
GetListItem	84
IsArgNull	85
T_IsList	86
T_IsNull	86

GetArg

Action Returns the value of the specified argument.

Syntax **GetArg**(*argnum*, *type*)

argnum The number of a 4Test argument.

type One of the following:

Type Value	Explanation
fValue	int (4Test boolean) value.
lValue	long (4Test integer) value.
dblValue	double (4Test real) value.
pszValue	string value.
List	list value.

Returns `pArgs->pData[argnum].Value.type`

Example The following code returns the integer value of the third 4Test argument

```
GetArg(2, lValue)
```

Notes GetArg retrieves the value of type *type* from the argument *argnum*. This macro requires that the argument to your extension function be called “pArgs”.

While arguments in 4Test are 1-based (that is, the first argument is argument number 1), in C all arrays are 0-based (that is, the first argument is argument number 0).

See also [“GetArgOpt” on page 83](#)

GetArgOpt

Action Retrieves an optional argument or its default value.

Syntax **GetArgOpt**(int *argnum*, **TYPE** *type*, *default*)

argnum The number of a 4Test argument.

type One of the following:

Type Value	Explanation
fValue	int (4Test boolean) value.
lValue	long (4Test integer) value.
dblValue	double (4Test real) value.
pszValue	string value.
List	list value.

default The default value of the argument.

Returns `(T_IsNull (pArgs->pData[argnum].Type) ? (default) : pArgs->pData[argnum].Value.type)`

Example The following code returns the integer value of the third 4Test argument, or returns 10 if the argument was not provided by the user. This macro requires that the argument to your extension function be called “pArgs”.

```
GetArgOpt(2, lValue, 10)
```

Notes GetArgOpt retrieves the value of type *type* from the optional argument *argnum*. If the argument *argnum* was not provided, GetArgOpt returns the value provided in *default*.

Note While arguments in 4Test are 1-based (that is, the first argument is argument number 1), in C all arrays are 0-based (that is, the first argument is argument number 0).

See also [“GetArg” on page 82](#)

GetArgType

Action Returns the 4test data type of a 4Test argument.

Syntax **GetArgType**(*argnum*)
argnum The number of a 4Test argument.

Returns *pArgs*->*pData*[**argnum**].Type

Example The following returns the 4Test data type of the third 4Test argument.
`GetArgType(2)`

Notes `GetArgType` returns the 4Test data type of the 4Test argument *argnum*. For more information, see [“4Test data types” on page 21](#) and [“Parameter attributes” on page 22](#). This macro requires that the argument to your extension function be called “pArgs”.

Note While arguments in 4Test are 1-based (that is, the first argument is argument number 1), in C all arrays are 0-based (that is, the first argument is argument number 0).

See also `IsArgList`, [“IsArgNull” on page 85](#)

GetListItem

Action Returns a specified item from a 4Test list.

Syntax **GetListItem**(*list*, *item*, *type*)
list A 4Test list.
item The number of the item in the list.
type One of the values in the following table:

Type Value	Explanation
fValue	int (4Test boolean) value.
lValue	long (4Test integer) value.
dblValue	double (4Test real) value.
pszValue	string value.
List	list value.

Returns	<code>(<i>list</i>).pData[<i>item</i>].Value.<i>type</i></code>
Example	The following code retrieves the integer value of the third item in the list which is the first 4Test argument: <code>GetListItem(GetArg(0, List), 2, lValue)</code>
Notes	GetListItem retrieves the item <i>item</i> of type <i>type</i> from the list <i>list</i> . The list can be retrieved from the ARGV structure using the GetArg macro. Note While lists in 4Test are 1-based (that is, the first item is item number 1), in C all arrays are 0-based (that is, the first item is item number 0).
See also	“GetArg” on page 82

IsArgNull

Action	Determines if an argument is NULL.
Syntax	IsArgNull (<i>argnum</i>) <i>argnum</i> The number of a 4Test argument.
Returns	TRUE if the argument is NULL (that is, its type is T_NULL) or FALSE if it is not.
Example	The following returns TRUE if the second argument to the 4Test method is NULL. <code>IsArgNull(1)</code>
Notes	IsArgNull returns TRUE if the 4Test data type of the 4Test argument <i>argnum</i> is NULL or FALSE if it is not. This macro requires that the argument to your extension function be called “pArgs”. Note While arguments in 4Test are 1-based (that is, the first argument is argument number 1), in C all arrays are 0-based (that is, the first argument is argument number 0).
See also	“T_IsNull” on page 86

T_IsList

Action	Determines if a 4Test data type is a list.
Syntax	T_IsList(TYPE <i>type</i>) <i>type</i> A 4Test data type.
Returns	TRUE if the 4Test data type is a list or FALSE if it is not.
Example	The following will return TRUE if the third 4Test argument is a list. <pre>T_IsList (pArgs->pData[2].Type)</pre>
Notes	T_IsList returns TRUE if the 4Test data type <i>type</i> is a list or FALSE if it is not. For more information about 4Test data types, see “4Test data types” on page 21 . This macro requires that the argument to your extension function be called “pArgs”.
See also	“GetArgType” on page 84

T_IsNull

Action	Determines if a 4Test data type is NULL.
Syntax	T_IsNull(TYPE <i>type</i>) <i>type</i> A 4Test data type
Returns	TRUE if the 4Test data type is NULL or FALSE if it is not.
Example	The following will return TRUE if the third 4Test argument is NULL. <pre>T_IsNull (pArgs->pData[2].Type)</pre>
Notes	T_IsNull returns TRUE if the 4Test data type <i>type</i> is NULL. For more information about 4Test data types, see “4Test data types” on page 21 . This macro requires that the argument to your extension function be called “pArgs”.
See also	“GetArgType” on page 84 , “IsArgNull” on page 85

Index

Numerics

4Test errors, raising 61

B

Boolean value, returning 70

C

class functions

 registering 63

 routing 76

 unregistering 82

clicking the mouse 54

ClickMouse function 54

E

errors, raising 9, 61

event generation 9, 54, 57, 58, 60, 67, 68, 80

exceptions, raising 9, 61

extensions

 initializing 56

 registering 56

 unregistering 79

F

functions

 QAP_ClickMouse 54

 QAP_Initialize 56

 QAP_MoveMouse 57

 QAP_PressKeys 58

 QAP_PressMouse 60

 QAP_RaiseError 61

 QAP_RegisterClassFun 63

 QAP_RegisterWindowFun 65

 QAP_ReleaseKeys 67

 QAP_ReleaseMouse 68

 QAP_ReturnBoolean 70

 QAP_ReturnInteger 71

 QAP_ReturnListClose 71

 QAP_ReturnListOpen 72

 QAP_ReturnNull 73

 QAP_ReturnReal 73

 QAP_ReturnString 74

 QAP_Terminate 79

 QAP_TypeKeys 80

 QAP_UnregisterClassFun 82

 QAP_UnregisterWindowFun 84

 registering 63, 65

 RouteAllClassFun 76

 RouteAllWindowsFun 77

 routing 76, 77

 unregistering 82, 84

G

generating events 9, 54, 57, 58, 60, 67, 68, 80

GetArg macro 88

GetArgOpt macro 89

GetArgType macro 90

GetListItem macro 90

H

handles to windows 9

I

Initialize function 56

initializing extensions 56

instance functions

 registering 65

 routing 77

 unregister 84

integer value, returning 71

IsArgNull macro 91

K

keyboard events 9

 generating 67

keys

 pressing 58

 releasing 67

 typing 80

L

lists, returning 71, 72

M

macros

- GetArg 88
- GetArgOpt 89
- GetArgType 90
- GetListItem 90
- IsArgNull 91
- T_IsList 92
- T_IsNull 92

mouse

- clicking 54
- moving 57
- pressing 60
- releasing 68

mouse events 9

- generating 54, 57, 58, 60, 68, 80

- MoveMouse function 57

- moving the mouse 57

N

- null value, returning 73

P

- pressing keys 58

- pressing the mouse 60

- PressKeys function 58

- PressMouse function 60

Q

- QAP_ClickMouse 54

- QAP_Initialize 56

- QAP_MoveMouse 57

- QAP_PressKeys 58

- QAP_PressMouse 60

- QAP_RaiseError 61

- QAP_RegisterClassFun 63

- QAP_RegisterWindowFun 65

- QAP_ReleaseKeys 67

- QAP_ReleaseMouse 68

- QAP_ReturnBoolean 70

- QAP_ReturnInteger 71

- QAP_ReturnListClose 71

- QAP_ReturnListOpen 72

- QAP_ReturnNull 73

- QAP_ReturnReal 73

- QAP_ReturnString 74

- QAP_RouteAllClassFun 76

- QAP_RouteAllWindowsFun 77

- QAP_Terminate 79

- QAP_TypeKeys 80

- QAP_UnregisterClassFun 82

- QAP_UnregisterWindowFun 84

R

- RaiseError function 61

- raising errors 9, 61

- raising exceptions 9, 61

- real value, returning 73

- records, returning 71, 72

- register

 - class functions 63

 - extensions 56

 - window functions 65

- RegisterClassFun function 63

- RegisterWindowFun function 65

- ReleaseKeys function 67

- ReleaseMouse function 68

- releasing keys 67

- releasing the mouse 68

- ReturnBoolean function 70

- returning values

 - Boolean 70

 - integer 71

 - lists 71, 72

 - null 73

 - real 73

 - records 71, 72

 - string 74

- ReturnInteger function 71

- ReturnListClose function 71

- ReturnListOpen function 72

- ReturnNull function 73

- ReturnReal function 73

- ReturnString function 74

- RouteAllClassFun function 76

- RouteAllWindowsFun function 77

- routing

 - class functions 76

 - window functions 77

S

- string value, returning 74

T

- T_IsList macro 92

- T_IsNull macro 92

- Terminate function 79

- TypeKeys function 80

typing keys 80

U

unregister

 class functions 82

 extension 79

 window functions 84

UnregisterClassFun 82

UnregisterWindowFun 84

W

window functions

 registering 65

 routing 77

 unregistering 84

window handles 9

window information 9

