

WINDOWS 32 API..... 1

- WHAT IS API?..... 2**
 - DESIGN MODELS..... 2
 - RELEASE POLICIES..... 3
 - API EXAMPLES..... 3
- THE WINDOWS API 4**
- DLL (DYNAMIC LINK LIBRARY) FILES 4**
- FUNDAMENTALS OF A DLL..... 4**
- BACKGROUND..... 5**
- HISTORY 5**
- WINDOWS MESSAGES 6**
- WINDOWS HANDLES 7**
- OVERVIEW OF THE WINDOWS API COMPONENTS 7**
 - BASE SERVICES 7
 - GRAPHICS DEVICE INTERFACE..... 7
 - USER INTERFACE 8
 - COMMON DIALOG BOX LIBRARY 8
 - COMMON CONTROL LIBRARY 8
 - WINDOWS SHELL..... 8
 - NETWORK SERVICES..... 9
 - WEB APIS..... 9
 - MULTIMEDIA RELATED APIS 9
 - APIS FOR INTERACTION BETWEEN PROGRAMS..... 10
- WINDOWS API TOOLS 11**
 - API VIEWER 11
- THE EXTERN OBJECT 12**
 - LIMITATIONS USING THE EXTERN OBJECT..... 12
 - CONVERTING TO EXTERN.DECLARE..... 13
 - Example 1: GetPrivateProfileString Function..... 13
 - Example 2: GetServiceKeyName Function..... 15
 - HIGHLIGHTS FOR WIN32 API IN QUICKTEST 17
 - Passing Parameters to Win32 API functions..... 18
 - Callback Functions..... 19
 - Interacting with a custom dll file 19
- THE EXTERN OBJECT ERROR! BOOKMARK NOT DEFINED.**
- APPENDIX 13.A 19**
 - Declare Data Types..... 22

Windows 32 API

This tutorial is about how to add additional functionality to your applications by using the Windows Advanced Programming Interface.

Windows API (Application Programming Interfaces) is a set of predefined Windows functions used to control the appearance and behavior of every Windows element (from the outlook of the desktop window to the allocation of memory for a new process). Every user action causes the execution of several or more **API**

function telling Windows what's happened.

It is something like the native code of **Windows**. Other languages just act as a shell to provide an automated and easier way to access **APIs**.

QuickTest supports a **VBScript** development language and its functionality can be extended in two ways. The first way, is by using custom controls. These are software components that are easily incorporated into **QTP/VBS** scripts. There are custom controls for almost any task imaginable, including numerical analysis, speech recognition, image and document support. Custom controls are called Ocx's.

The second way, **QuickTest** can be extended is through the **Windows** Application Programming Interface (**API**), using the **Extern object**. **API** functions are not supported in **VBScript**.

What is API?

An application programming interface (**API**) is a source code interface that a computer system or program library provides in order to support requests for services to be made of it by a computer program.

An API differs from an Application Binary Interface in that it is specified in terms of a programming language that can be compiled when an application is built, rather than an explicit low level description of how data is laid out in memory.

The software that provides the functionality described by an API is said to be an implementation of the API. The API itself is abstract, in that it specifies an interface and does not get involved with implementation details.

An API is often a part of a software development kit (SDK).
The term API is used in two related senses:

- A coherent interface consisting of several classes or several sets of related functions or procedures.
- A single entry point such as a method, function or procedure.

In general terms, an **API** is system software for an operating system or environment, which consists of a standardized set of functions and procedures. Programmers can call these functions and procedures from their programs to gain extra functionality. Because programmers do not have to write this code themselves, they save time. The system also provides a standard and well documented way of working

Design Models

There are various design models for APIs. Interfaces intended for the fastest execution often consist of sets of functions, procedures, variables and data structures. However, other models exist as well - such as the interpreter used to evaluate expressions in *ECMAScript/JavaScript* or in the abstraction layer - which relieve the programmer from needing to know how the functions of the **API** relate to the lower levels of abstraction. This makes it possible to redesign or improve the functions within the **API** without breaking code that relies on it.

Some APIs, such as the ones standard to an operating system, are implemented

as separate code libraries that are distributed with the operating system. Others require software publishers to integrate the API functionality directly into the application. This forms another distinction in the examples above. Microsoft Windows APIs come with the operating system for anyone to use. Software for embedded systems such as video game consoles generally falls into the application-integrated category. While an official PlayStation API document may be interesting to read, it is of little use without its corresponding implementation, in the form of a separate library or software development kit.

An API that does not require royalties for access and usage is called "open".^[1] Although usually authoritative "reference implementations" exist for an API (such as Microsoft Windows for the Win32 API), there is nothing that prevents the creation of additional implementations. For example, most of the **Win32 API** can be provided under a **UNIX** system using software called **Wine**.

Release Policies

Two general lines of API publishing policies:

Some companies zealously guard information on their APIs from general public consumption. For example, Sony used to make its official PlayStation 2 API available only to licensed PlayStation developers. This enabled Sony to control who wrote PlayStation 2 games. Such control can have quality control benefits and potential license revenue.

Some companies make their APIs freely available. For example, Microsoft makes most of its API information public, so that software will be written for the Windows platform.

Companies base their choice of publishing policy on maximizing benefit to them.

API Examples

- The PC BIOS call interface
- Single UNIX Specification (SUS)
- Microsoft Win32 API
- Java Platform, Enterprise Edition APIs
- ASPI for SCSI device interfacing
- Carbon and Cocoa for the Macintosh OS
- OpenGL cross-platform API
- DirectX for Microsoft Windows
- Simple DirectMedia Layer (SDL)
- Universal Home API
- LDAP Application Program Interface
- svalglib for Linux and FreeBSD
- Google Maps API
- Wikipedia API
- Webmashup.com The Open Directory for Mashups & Web 2.0 APIs

The Windows API

The API functions reside in DLLs (like User32.dll, GDI32.dll, Shell32.dll, ...) in the Windows system directory.

The **Windows API** is the name given by Microsoft to the core set of application programming interfaces available in the **Microsoft** Windows operating systems. It is designed for usage by C/C++ programs and is the most direct way to interact with a **Windows** system for software applications. Lower level access to a Windows system

A **Software Development Kit (SDK)** is available for **Windows**, which provides documentation and tools to enable developers to create software using the **Windows API** and associated **Windows** technologies.

The **Windows API** is the set of functions and procedures available in **Windows**. They can be used by all sorts of programmers. Mercury added the **API** access interface including those working in **QuickTest**.

The most often required functionality of the **Windows API** already exists in the **VBScript** basic development environment. Most of the **API** functions and procedures that have not been included in **VBScript** itself can still be called from the **QuickTest** application. The small percentage that cannot, are rarely required.

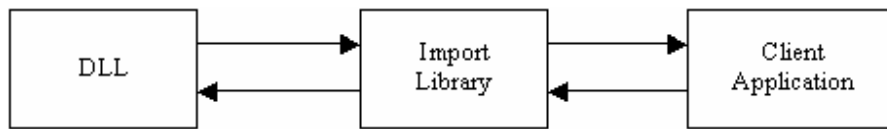
DLL (Dynamic Link Library) Files

Dynamic-link library (**DLL**), also known as dynamic link library (without the hyphen), is Microsoft's implementation of the shared library concept in the Microsoft Windows operating systems. These libraries usually have the file extension **DLL**, **OCX** (for libraries containing ActiveX controls), or DRV (for legacy system drivers).

Dynamic Link Library, a library of executable functions or data that can be used by a Windows application. Typically, a **DLL** provides one or more particular functions and a program accesses the functions by creating either a static or dynamic link to the **DLL**. A static link remains constant during program execution while a dynamic link is created by the program as needed. **DLLs** can also contain just data. **DLL** files usually end with the extension .dll, .exe., drv, or .fon. A **DLL** can be used by several applications at the same time. Some **DLLs** are provided with the Windows operating system and available for any Windows application. Other **DLLs** are written for a particular application and are loaded with the application.

Fundamentals of a DLL

We have mentioned that a **DLL** is created as a project that contains at least one source file and this source file should present an entry-point. After creating the **DLL**, you will build and distribute it so other programs can use it. When building it, you must create a library file that will accompany the **DLL**. This library file will be used by other programs to import what is available in the **DLL**:



When the import library is created, it contains information about where each available function is included in the **DLL** and can locate it. When an application needs to use a function contained in the **DLL**, it presents its request to the import library. The import library checks the **DLL** for that function. If the function exists, the client program can use it. If it doesn't, the library communicates this to the application and the application presents an error.

Background

The original purpose for **DLLs** was saving both disk space and memory required for applications by storing it locally on the hard drive. In a conventional non-shared library, sections of code are simply added to the calling program. If two programs call the same routine, that code would be duplicated. Instead, any code which many applications share could be separated into a **DLL** which only exists as a single disk file and a single instance in memory. Extensive use of **DLLs** allowed early versions of **Windows** to work under tight memory conditions.

DLLs provide the standard benefits of shared libraries, such as modularity. Modularity allows changes to be made to code and data in a single self-contained **DLL** shared by several applications without any change to the applications themselves. This basic form of modularity allows for relatively compact patches and service packs for large applications, such as **Microsoft Office**, **Microsoft Visual Studio**, and even **Microsoft Windows** itself.

Another benefit of the modularity is the use of generic interfaces for plug-ins. A single interface may be developed which allows old as well as new modules to be integrated seamlessly at run-time into pre-existing applications, without any modification to the application itself. This concept of dynamic extensibility is taken to the extreme with **ActiveX**.

With this many benefits, using **DLLs** also has a drawback: the **DLL** hell, when several applications conflict on which version of a shared **DLL** library is to be used. Such conflicts can usually be resolved by placing the different versions of the problem **DLL** into the applications' folders, rather than a system-wide folder; however, this also nullifies the savings provided by using shared **DLLs**. Currently, **Microsoft .NET** is targeted as a solution to the problem of **DLL** hell by allowing side-by-side coexistence of different versions of a same shared library. With modern computers which have plenty of disk space and memory, it can be a reasonable approach.

History

The **Windows API** has always exposed a large part of the underlying structure of the various Windows systems for which it has been built to the programmer. This has had the advantage of giving Windows programmers a great deal of flexibility and power over their applications. However, it also has given Windows applications a great deal of responsibility in handling various low-level, sometimes tedious,

operations that are associated with a **Graphical user interface**.

A **hello world** program is a frequently used programming example, usually designed to show the easiest possible application on a system that can actually do something (i.e. print a line that says "Hello World").

Over the years, various changes and additions were made to the **Windows Operating System**, and the **Windows API** changed and grew to reflect this. The **Windows API** for Windows 1.0 supported fewer than 450 function calls, where in modern versions of the **Windows API** there are thousands. However, in general, the interface remained fairly consistent, and an old Windows 1.0 application will still look familiar to a programmer who is used to the modern **Windows API**. [12]

A large emphasis has been put by **Microsoft** on maintaining software backwards compatibility. To achieve this, **Microsoft** sometime even went as far as supporting software that was using the **API** in an undocumented or even (programmatically) illegal way.

One of the largest changes the **Windows API** underwent was the transition from **Win16** (shipped in Windows 3.1 and older) to **Win32** (Windows NT and Windows 95 and up). While **Win32** was originally introduced with Windows NT 3.1 and Win32s allowed usage of a **Win32** subset before Windows 95, it was not until Windows 95 that many applications began being ported to **Win32**. To ease the transition, in Windows 95, both for external developers and for Microsoft itself, a complex scheme of **API** thunks was used that could allow 32 bit code to call into 16 bit code and (in limited cases) vice-versa. So-called flat thunks allowed 32 bit code to call into 16 bit libraries, and the scheme was used extensively inside Windows 95 to avoid porting the whole OS to **Win32** itself in one chunk. In Windows NT, the OS was pure 32-bit (except the parts for compatibility with 16-bit applications) and the only thunk available was generic thunks which only thunks from **Win16** to **Win32** and worked in Windows 95 too. The **Platform SDK** shipped with a compiler that could produce the code necessary for these thunks.

Windows Messages

Messages are the basic way **Windows** tells your program that some kind of input has occurred and you must process it. A message to your form is sent when the user clicks on a button, moves the mouse over it or types text in a textbox. All messages are sent along with four parameters

- A window handle, a message identifier and two 32-bit (Long) values. The window handle contains the handle of the window the message is going to.
- The identifier is actually the type of input occurred (click, mousemove).
- Two value specify an additional information for the message (like where is the mouse cursor when the mouse is been moved).

But, when messages are sent to you, why you don't see them, looks like someone is stealing your mail. And before you get angry enough, let me tell you.

The theft is actually Your Application. But he does not steal your mail, but instead read it for you and give you just the most important in a better look (with some information hidden from time to time). This better look is the events you write code for.

So, when the user moves the mouse over your form, Windows sends

WM_MOUSEMOVE to your window, Your application get the message and its parameters and executes the code you've entered for **Button_MouseMove** event.

Something else that needs to be said: You can send messages to your own window or to another one yourself. You just call **SendMessage** or **PostMessage** (**SendMessage** will cause the window to process the message immediately and **PostMessage** will post it onto a queue, called message queue, after any other messages waiting to be processed (it will return after the message is processed, i.e. with some delay)). You must specify the window handle to send the message to, the message and the two 32-bit values.

Windows Handles

A window handle (usually shortened to hWnd) is a unique identifier that Windows assigns to each window created. By window in this case we are referring to everything from command buttons and textboxes, to dialog boxes and full windows.

The window handle is used in **APIs** as the sole method of identifying a window. It is a **Long** (4 byte) value and you can get the handle for forms and almost all controls.

Windows identifies every form, control, menu, button, and menu item or whatever you can think of by its handle. When your application is run, every control on it is assigned a handle which is used later to separate the button from the rest of the controls. If you want to perform any operation on the button through an **API** you must use this handle. Where to get it from? Well **API** has provided an **hWnd** property for all controls that have handles in **Windows**.

Overview of the Windows API Components

Base Services

Provide access to the fundamental resources available to a Windows system. Included are things like file systems, devices, processes and threads, access to the Windows registry, and error handling. These functions reside in kernel.exe, krnl286.exe or krnl386.exe files on 16-bit Windows, and kernel32.dll and advapi32.dll on 32-bit Windows.

For more information:

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/winprog/winprog/base_services.asp

Graphics Device Interface

Provide the functionality for outputting graphical content to monitors, printers and other output devices. It resides in gdi.exe on 16-bit Windows, and gdi32.dll on 32-bit Windows.

For more information:

<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/winprog/winprog/gdi32.dll>

[us/winprog/winprog/graphics_device_interface.asp](http://msdn.microsoft.com/library/default.asp?url=/library/en-us/winprog/winprog/graphics_device_interface.asp)

User Interface

Provides the functionality to create and manage screen windows and most basic controls, such as buttons and scrollbars, receive mouse and keyboard input, and other functionality associated with the **GUI** part of **Windows**. This functional unit resides in user.exe on 16-bit Windows, and user32.dll on 32-bit Windows. Since **Windows XP** versions, the basic controls reside in comctl32.dll, together with the common controls (**Common Control Library**).

For more information:

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/winprog/winprog/user_interface.asp

Common Dialog Box Library

Provides applications the standard dialog boxes for opening and saving files, choosing color and font, etc. The library resides in a file called comdlg.dll on 16-bit Windows, and comdlg32.dll on 32-bit Windows. It is grouped under the User Interface category of the **API**.

For more information:

<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/winui/winui/windowsuserinterface/userinput/commondialogboxlibrary.asp>

Common Control Library

Gives applications access to some advanced controls provided by the operating system. These include things like status bars, progress bars, toolbars and tabs. The library resides in a **DLL** file called commctrl.dll on 16-bit Windows, and comctl32.dll on 32-bit Windows. It is grouped under the User Interface category of the **API**.

For more information:

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/winprog/winprog/common_control_library.asp

Windows Shell

Component of the Windows **API** allows applications to access the functionality provided by the operating system shell, as well as change and enhance it. The component resides in shell.dll on 16-bit Windows, and shell32.dll and later in Windows 95 shlwapi.dll on 32-bit Windows. It is grouped under the User Interface category of the **API**.

For more information:

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/winprog/winprog/windows_shell.asp and
<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/winprog/winprog/shlwapi.asp>

[us/shellcc/platform/shell/programmersguide/shell_intro.asp](http://msdn.microsoft.com/library/default.asp?url=/library/en-us/shellcc/platform/shell/programmersguide/shell_intro.asp)

Network Services

Give access to the various networking capabilities of the operating system. Its sub-components include NetBIOS, Winsock, NetDDE, RPC and many others.

For more information:

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/winprog/winprog/network_services.asp

Web APIs

The Internet Explorer web browser also exposes many APIs that are often used by applications, and as such could be considered a part of the Windows API. Internet Explorer has been an integrated component of the operating system since Windows 98, and provides web related services to applications. The integration will stop with Windows Vista. Specifically, it provides:

- An embeddable web browser control, contained in shdocvw.dll and mshtml.dll.
- The URL monikers service, held in urlmon.dll, which provides COM objects to applications for resolving URLs. Applications can also provide their own URL handlers for others to use.
- A library for assisting with multi-language and international text support (mlang.dll).
- DirectX Transforms, a set of image filter components.
- XML support (the MSXML components).
- Access to the Windows Address Book.

Multimedia related APIs

Microsoft has provided the (DirectX) set of APIs as part of every Windows installation since Windows 95 OSR2. DirectX provides a loosely related set of multimedia and gaming services, including:

- **Direct3D** as an alternative to OpenGL for access to 3D acceleration hardware.
- **DirectDraw** for hardware accelerated access to the 2D framebuffer. As of DirectX 9, this component has been deprecated in favor of Direct3D, which provides more general high-performance graphics functionality (2D rendering, after all, is really just a subset of 3D rendering).
- **DirectSound** for low level hardware accelerated sound card access.
- **DirectInput** for communication with input devices such as joysticks and gamepads.
- **DirectPlay** as a multiplayer gaming infrastructure. This component has been deprecated as of DirectX 9 and Microsoft no longer recommends its use for game development.
- **DirectShow** which builds and runs generic multimedia pipelines. It is

comparable to the GStreamer framework and is often used to render in-game videos and build media players (Windows Media Player is based upon it).

DirectShow is no longer recommended for game development.

- **DirectMusic**

APIs for interaction between programs

The Windows API mostly concerns itself with the interaction between the Operating System and an application. For communication between the different Windows applications among themselves, Microsoft has developed a series of technologies alongside the main Windows API. This started out with Dynamic Data Exchange (**DDE**), which was superseded by Object Linking and Embedding (**OLE**) and later by the Component Object Model (**COM**).

Windows API Tools

API Viewer

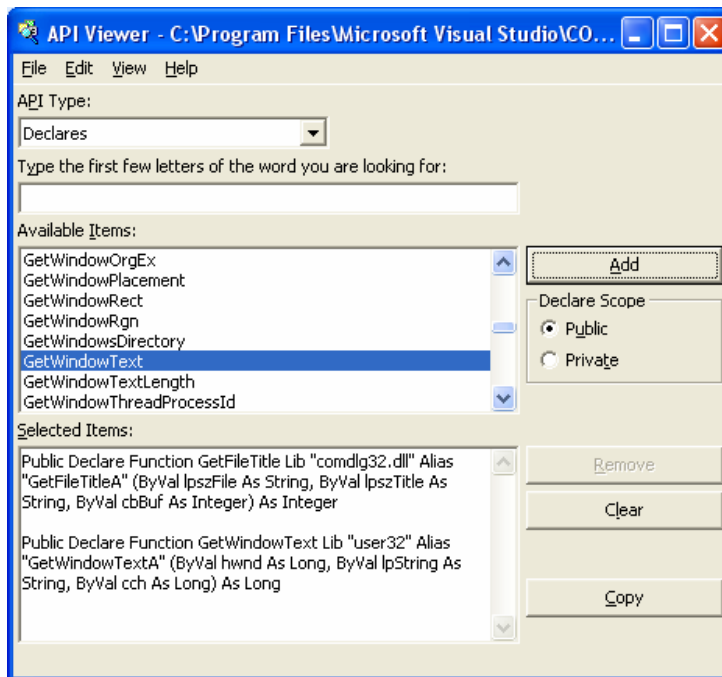


Figure 1 – Visual Studio API Viewer

If you have installed Microsoft Visual Studio, there is a folder named "Common\Tools\Winapi" This folder contains the text file WIN32API.TXT, along with other files. The WIN32API.TXT file holds the "Constants," "Declares" and "Types" for the 32-bit versions of the **Windows API** functions. The file APILOD.TXT in the same folder contains more information.

Double-clicking WIN32API.TXT loads it using WordPad - it is too large to open using Notepad. But there's a better and more efficient way of getting **API** information into your application than using the "Copy" and "Paste" functions on the WIN32API.TXT file.

Double-clicking "APILOD32.EXE" loads the **API Viewer** or In Start-> Programs -> Microsoft Visual Studio -> Tools -> API text Viewer. With it you can look at the Constants, Declares, and Types of the **API**. It works as follows:

Click on the "File" menu, then the menu option "Load Text File ... "

Select the WIN32API.TXT item in the list box and click the "Open" command button. You'll be asked if you would like to convert the API file to a database. If you answer yes, the information will load more quickly, so click the "Yes" command button. Next you will be asked to save it, with the recommendation to save it as WIN32API.MDB. Accept this recommendation by clicking the "Save" command button. You'll only have to do this the first time.

The next time you use the API Viewer you can select the "File" menu option. Then the "Load Database File ..." option. From the list that appears you select the item "WIN32API.MDB".

Once the database file information has been loaded, use the **API Viewer** to select one of the categories of Constants, Declares, or Types - and the items available in each category will now be displayed.

The Problem, for **QuickTest** users that the declaration section has not the same syntax of the **Extern.Declare** method, but, you can use the **API Text Viewer** to help you how to define the extern declarations.

The Extern Object

Description

Enables you to declare calls to external procedures from an external dynamic-link library (DLL).

Syntax

```
Extern.Declare RetType, MethodName, LibName, Alias, ArgType(s)
```

Arguments

Parameter	Description
<i>RetType</i>	Data type of the value returned by the method. For available data types, see Declare Data Types in Table 1 on page 22
<i>MethodName</i>	A String that can be any valid procedure name.
<i>LibName</i>	A String that represents the name of the DLL or code resource that contains the declared procedure.
<i>Alias</i>	A String that represents the name of the procedure in the DLL or code resource. Note: DLL entry points are case sensitive. Note: If <i>Alias</i> is an empty string, <i>MethodName</i> is used as the <i>Alias</i> .
<i>ArgType(s)</i>	A list of data types representing the data types of the arguments that are passed to the procedure when it is called. For available data types, see Declare Data Types in Table 1 on page 22 Note: For out arguments, use the micByRef flag.

Limitations using the Extern Object

The QuickTest **Extern** object does not support API functions with Callback mechanism, and also does not support any function with a structure parameter, and a pointer to function parameter.

```
BOOL GetCursorInfo( PCURSORINFO pci);
```

The function uses a **PCURSORINFO**¹ parameter that is not supported by the extern function

```
BOOL CALLBACK EnumChildProc(HWND hwnd, LPARAM lParam);
```

The **EnumChildProc** function is an application-defined callback function used with the **EnumChildWindows** function.

Dani Vainstein

¹ Will be expanded on the COM Chapter.

```
BOOL EnumChildWindows(HWND hWndParent, WNDENUMPROC lpEnumFunc, LPARAM lParam);
```

The Parameter *lpEnumFunc* is a pointer to an application-defined callback function.

Converting to Extern.Declare

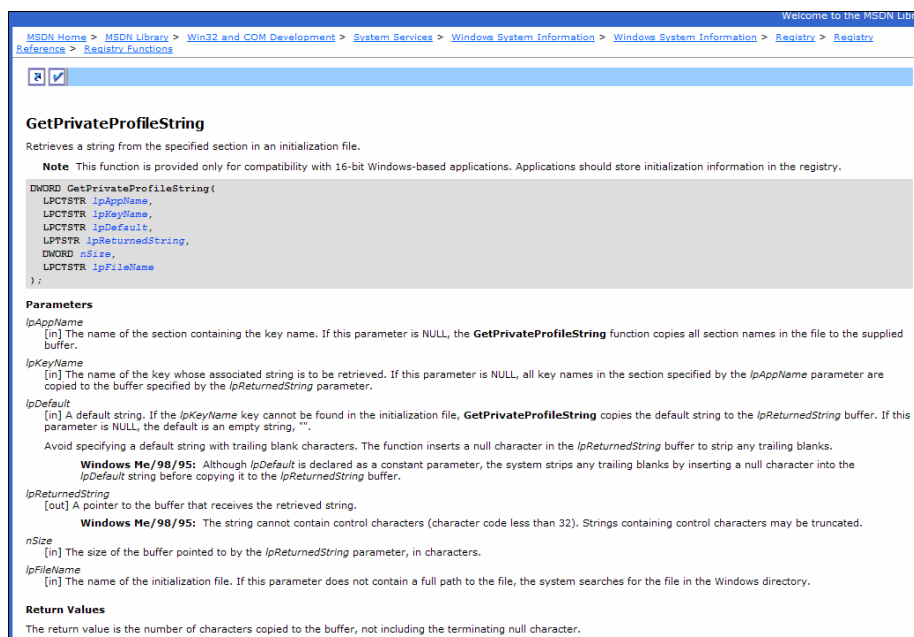
The **QuickTest Extern object** does not support API functions with Callback mechanism, and also it does not support any function with a structure parameter, and a pointer to function parameter.

As said before, there are hundreds of **Windows API** function. In most of the cases, you won't need the **API**, because there are many automation objects (ActiveX, ocx) that make us live easier to manipulate, a lot of **Windows API** functionality, without using **Windows API** directly (ADODB, EXCEL, MAPI, SHELL, WSH and more)

Well, we found the **Windows API** function that we need, what's next? **API** functions syntax and documentation can be found in **MSDN** and all over the internet. You should look for functions in **Visual Basic** style, or you can use the **API Text Viewer**.

Example 1: GetPrivateProfileString Function

The Function retrieves a string from the specified section in an initialization file (ini file)



The screenshot shows the MSDN documentation for the `GetPrivateProfileString` function. The breadcrumb trail is: MSDN Home > MSDN Library > Win32 and COM Development > System Services > Windows System Information > Windows System Information > Registry > Registry Reference > Registry Functions. The function signature is: `DWORD GetPrivateProfileString(LPCWSTR lpAppName, LPCWSTR lpKeyName, LPCWSTR lpDefault, LPWSTR lpReturnedString, DWORD nSize, LPCWSTR lpFileName);`. The documentation includes a note about compatibility with 16-bit applications, a list of parameters with their descriptions and Windows Me/98/95 specific details, and a return value description.

Figure 2 - GetPrivateProfileString MSDN Documentation

First, We need the syntax of the function:

```

DWORD GetPrivateProfileString(
    LPCTSTR lpAppName,
    LPCTSTR lpKeyName,
    LPCTSTR lpDefault,
    LPTSTR lpReturnedString,
    DWORD nSize,
    LPCTSTR lpFileName
);

```

We can see that the function returns a DWORD value, according to the Declare Data Types DWORD = micDWord, The Second argument is the name of the function to be used in QuickTest

```
Extern.Declare micDWord, "GetIniString"
```

The *LibName* argument can be find in the bottom of the MSDN function documentation

Client	Requires Windows Vista, Windows XP, Windows 2000 Professional, Windows 95.
Server	Requires Windows Server "Longhorn", Windows Server 2003.
Header	Declared in Winbase.h; include Windows.h.
Library	Use Kernel32.lib.
DLL	Requires Kernel32.dll.
Unicode	Implemented as GetPrivateProfileStringW (Unicode) and GetPrivateProfileStringA (ASCII). Windows Me/98/95 requires Microsoft Layer for Unicode.

Not the **DLL** section, Kernel32.dll.

```
Extern.Declare micDWord, "GetIniString", "kernel32.dll"
```

Now, The *Alias* argument. To retrieve the exactly alias argument you will need to search more over the internet, especially on **Visual Basic API** declarations. You can notice in the documentation that the function is called **WritePrivateProfileA** and also **GetPrivateProfileW**, where **A** means **ASCII** and **W** means wide-char (Unicode). When an **API** function have a string in one of the arguments, you need to use the **A** post-letter.

```
Extern.Declare micDWord, "GetIniString", "kernel32.dll", "GetPrivateProfileStringA"
```

Last, the argument list. You have to be attented on 2 things

1. The argument data type
2. The argument direction (in, out or in-out)

LPCTSTR means: **L**ong **P**ointer to a **C**onstant null-**T**erminated **STR**ing

LPTSTR means: **L**ong **P**ointer to a null-**T**erminated **STR**ing

means both are strings, but the difference between them is the direction

Parameters

lpAppName
[in] The name of the section containing the key name. If the buffer.

lpKeyName
[in] The name of the key whose associated string is to be copied to the buffer specified by the *lpReturnedString* parameter.

lpDefault
[in] A default string. If the *lpKeyName* key cannot be found and the *lpDefault* parameter is NULL, the default is an empty string, "".

Avoid specifying a default string with trailing blank characters.

Windows Me/98/95: Although *lpDefault* is declared as a string, it is a pointer to a buffer. Copy the *lpDefault* string before copying it to the *lpReturnedString* parameter.

lpReturnedString
[out] A pointer to the buffer that receives the retrieved string.

Windows Me/98/95: The string cannot contain characters that are not valid in a file name.

nSize
[in] The size of the buffer pointed to by the *lpReturnedString* parameter.

lpFileName
[in] The name of the initialization file. If this parameter does not exist, the function fails.

Return Values

The return value is the number of characters copied to the buffer. If the buffer is not large enough to hold the string, the return value is less than *nSize*.

If neither *lpAppName* nor *lpKeyName* is NULL and the supplied buffer is not large enough to hold the string, the return value is equal to *nSize* minus one.

If either *lpAppName* or *lpKeyName* is NULL and the supplied buffer is not large enough to hold the string, the return value is equal to *nSize* minus two.

Note *lpReturnedString* is [out] parameter (LPTSTR)

```
Extern.Declare micDWord, "GetIniString", "kernel32.dll", "GetPrivateProfileStringA", _
micString, micString, micString, micString + micByRef, micDWord, micString
```

In your code, you will use the function using the following syntax

```
Dim nBytes, sRetVal
nBytes = Extern.GetIniString ("Section", "Key", "", sRetVal, 256, "C:\Sample.ini")
MsgBox "Bytes Returned = " & nBytes
MsgBox sRetVal
```

Example 2: GetServiceKeyName Function

This function retrieves a key name from a Windows service name.

```
Public Declare Function GetServiceKeyName Lib "advapi32.dll" Alias
"GetServiceKeyNameA" (ByVal hSCManager As Long, ByVal lpDisplayName As
String, lpServiceName As String, lpCchBuffer As Long) As Long
```

The **QuickTest** Declare method for Extern object must start with the declaration.

```
Extern.Declare
```

the VB declaration we see the last statement As Long. That means that the function returns a Long data type, in the data types table **Long = micLong**

```
Extern.Declare micLong
```

The method name value can be any unique string. It is recommended to use the function name itself, to following coding standards.

```
Extern.Declare micLong, "GetServiceKeyName"
```

- The argument LibName In the VB declaration is the keyword after Lib. This

argument can be written in 3 different ways:

- "advapi32.dll" – This is a system dll file, will be recognized from anywhere in your application.
- "advapi32" – you also can remove the extension, because is a known system dll file.
- "C:\Windows\System32\advapi32.dll" – if the used dll, is not a system dll, and not stored under System32, you should write the full path.

```
Extern.Declare micLong, "GetServiceKeyName", "advapi32.dll"
```

The alias argument in can e found in VB declaration after the Alias keyword. The alias name is very important. Is the physical name of the function in the dll file. Sometimes this name changes to postfix 'A' (ANSI) 'W' (Unicode/WideChar). Pay attention to this name, is case sensitive. Any syntax mistake will raise a general error message at run-time. Usually functions that one of the arguments is a string, the Alias name changes.

```
Extern.Declare micLong, "GetServiceKeyName", "advapi32.dll", "GetServiceKeyNameA"
```

The argument(s) In the VB declaration, the arguments are all the parameters inside the parenthesis.

```
Extern.Declare micLong, "GetServiceKeyName", "advapi32.dll", "GetServiceKeyNameA", micLong, micString, micString, micLong
```

Notice the argument list. Near every parameter in the VB declaration there is a **ByVal** keyword, excluding the lpccchBuffer and lpServiceName parameters, they are empty; it means that the parameters are return values (**ByRef**). So, here is the final declaration:

```
Extern.Declare micLong, "GetServiceKeyName", "advapi32.dll", "GetServiceKeyNameA", micLong, micString, micString + micByRef, micLong + micByRef
```

Using the Function.

- The function activation must be after the function declaration!
- You can put your declaration one line before the usage, or in the top of the action
- You can put the declaration in an Init reusable action.
- You can put your declaration in a VBS resource file.

In the following example we are going to retrieve the Key name of the **Terminal Services** Service. Be sure that you have the Service; otherwise choose another service from Control Panel->Administrative Tools->Services.

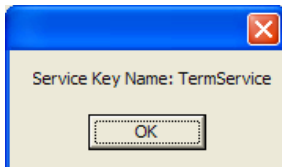
To activate this particular function we need the handle of the service manager, and then we need to close the service manager handle (release the memory)

```
Option Explicit
Extern.Declare micLong, "OpenSCManager", "advapi32", "OpenSCManagerA", _
    micString, micString, micLong
Extern.Declare micLong, "CloseServiceHandle", "advapi32.dll", "", micLong
Extern.Declare micLong, "GetServiceKeyName", "advapi32", "GetServiceKeyNameA", _
    micLong, micString, micString + micByRef, micLong + micByRef
Private Const GENERIC_READ = &H80000000
Dim hSCManager
Dim sKeyName
Dim nRet, nBuffer
```

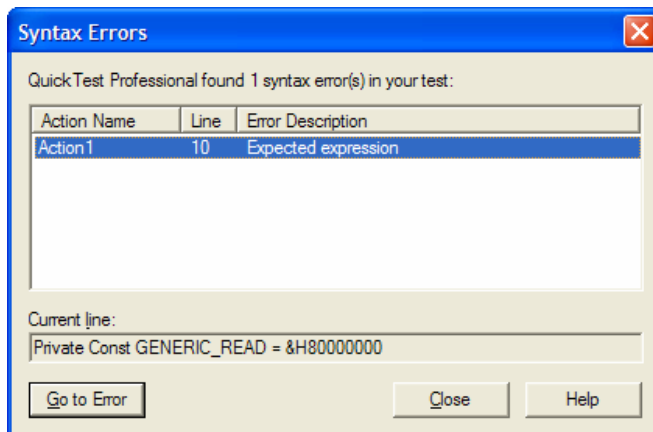
```

hSCManager = Extern.OpenSCManager(vbNullString, vbNullString, GENERIC_READ)
If hSCManager > 0 Then
    nBuffer = 0
    Extern.GetServiceKeyName hSCManager,"Terminal Services", sKeyName, nBuffer
    MsgBox "Buffer size required: " & nBuffer
    nRet=Extern.GetServiceKeyName( hSCManager,"Terminal Services", sKeyName, nBuffer)
    If nRet Then
        MsgBox "Service Key Name is: " & sKeyName
        nRet = Extern.CloseServiceHandle(hSCManager)
    End If
End If

```



The constant declaration was written in is hexadecimal value. If you move to the keyword view in QTP or make a syntax check you will get the following error message:



This is a QTP bug. Is a legal statement in VBS, running the script will not raise a run-time error

Highlights for Win32 API in QuickTest

- You should notice that the BOOL type (Boolean) evaluates to Integer and not Boolean. So, 0 (zero) refers to **False** and any other value to **True**.
- HWND, HDC, HMENU, etc. - and other types like these. All of them begin with **H** and means → HANDLE, for handles for different type of objects. For example HBITMAP is a bitmap handle; HBRUSH is a brush handle and so on. They all evaluate to Long and should be passes ByVal.
- Notice also that LPVOID is declared as variable As **Any**. Some messages contain parameters declared as **Any**. It means this parameter can be a variety of types (you may pass an integer, a string, or a user-defined type, or else).
- Some types begin with **LP**. It is an abbreviation of **Long Pointer** to. So **LPWORD** is actually a memory location where the data is stored. No, you

won't have to call a function to get this address. When you pass your argument ByRef (the default) you actually pass its address. The thing to remember here is that, if your parameter type begins with **LP** - you should pass it ByRef. By the way, LPARAM is not a pointer. You must pass the actual value here, so it is passed ByVal.

- There is also some strange type NULL. Just choose a way you will pass it when needed. In most of the cases use **ByVal 0&** for numbers, **Null** or **vbNull**. and for strings use **vbNullString**, for a single char use **vbNullChar**.
- VOID is used for functions return value to specify that there is no such value. API does not have Subs so this is the way it implements them. Just remember - if the return value is VOID use micVoid.
- Structures or Complex data types. There is no way to use those data types in QuickTest via Extern object.
- Well, if the return value is a structure, you can use micUnknown, and manipulate it somehow in your code, but, using API, functions does not return structures, you must pass the structure ByRef. And here is the problem, the Extern object can't do this, **VBScript** is limited (no API support at all) so, is no way to build a Structure inside **QuickTest** or VBS.
- For Example, There is a very useful API function GetCursorPos, The function returns the position of the cursor via the POINT structure.
- The solution for this problem in Using User-Custom COM classes.

Passing Parameters to Win32 API functions

Of course you know how to pass parameters, just put it in the function call and it's done! Well there are some details you should be aware of when passing parameters to API functions.

ByVal or **ByRef**. Usually you don't have to bother about these keywords as VB API Text Viewer declares the function parameters as API wants them and when you just enter your value, it is passed as is declared.

Generally, when a value is passed **ByVal**, the actual value is passed directly to the function, and when passed **ByRef**, the address (pointer) of the value is passed.

Passing strings to **API** function isn't difficult too. The **API** expects the address of the first character of the string and reads ahead of this address till it reaches a **Null** character. Sounds bad, but this the way the OS actually handles strings. The only thing to remember is always to pass the String **ByRef**.

The situation is slightly different when you expect some information to be returned by the function. Here is the declaration of **GetComputerName** API functions:

```
Extern.Declare micLong, "GetComputerName", _
    "kernel32.dll", "GetComputerNameA", micString + micByRef, micLong + micByRef
```

The first parameter is a long pointer to string, and the second the length of the string. If you just declare a variable of Buffer Size and pass it to this function, an error occurs. So, you need to initialize the buffer size first. Here is how to get the computer name:

```
Option Explicit
Extern.Declare micLong, "GetComputerName", "kernel32.dll", "GetComputerNameA", _
    micString + micByRef , micLong + micByref
```

```
Dim sBuffer
Dim nRet, nBufferSize
nBufferSize = 255
nRet = Extern.GetComputerName(sBuffer, nBufferSize)
```

Some functions also expect arrays. Here is an example. The last two parameter are arrays of Long. To pass an array to a function, you pass just the first element. Here is a sample code:

```
Option Explicit
Extern.Declare micLong, "SetSysColors", user32, "SetSysColors", _
    micLong, micLong+micByref, micLong+micByref
Private Const COLOR_ACTIVECAPTION = 2
Private Const COLOR_INACTIVECAPTION = 3
Private Const COLOR_CAPTIONTEXT = 9
Private Const COLOR_INACTIVECAPTIONTEXT = 19
Dim SysColor(3), ColorValues(3)
Dim nRet
SysColor(0) = COLOR_ACTIVECAPTION
SysColor(1) = COLOR_INACTIVECAPTION
SysColor(2) = COLOR_CAPTIONTEXT
SysColor(3) = COLOR_INACTIVECAPTIONTEXT
ColorValues(0) = RGB(58, 158, 58) 'dark green
ColorValues(1) = RGB(93, 193, 93) 'light green
ColorValues(2) = 0 'black
ColorValues(3) = RGB(126, 126, 126) 'gray
nRet = Extern.SetSysColors(4, SysColor(0), ColorValues(0))
```

Callback Functions

A **callback** is a function you write and tell Windows to call for some reason. You create your own function with a specified number and type of parameters, then tell Windows that this function should be called for some reason and its parameters filled with some info you need. Then Windows calls your function, you handle the parameters and exit from the function returning some kind of value. A typical use of callbacks is for receiving a continuous stream of data from Windows. This topic is also not supported by QuickTest.

Interacting with a custom dll file

You can also interact with your customs DLL's, your tested application DLL's. Whenever they written in VB, C or C++.

Microsoft C Runtime Library has the file name msvcrt.dll.

```
Extern.Declare micLong, "abs", "C:\RTO\msvcrt.dll", "abs", micLong
Msgbox Extern.abs(-55)
```

Interacting Between Win32 API and QuickTest

A very nice and usefull feature (for me) is to automatically highlight an object before accessing it.

For example I want to Click a button in my application, using **QTP**.

For me, to implement the code is very simple

```
Window( "W" ).WinButton( "B" ).Click
```

For me is 1 line of code, but, telling you the truth, this function does much more.

- Checks if The object exists.
- Check if the object is enabled.
- Highlight the object
- Click

First I use the RegisterUserFunc feature to override the common object method

Inside the overridden function I perform an object.Exist and object.enabled query

The I call to some win32 API for the highlight feature , and last clicking the button.

More for TextEdit, List, WebEdit, WebList I also add a synchronization point for the expected result.

When highlighting every object before QTP access it, will downgrade the performance.

Your script will be slower, but I think it worth it. You can visually track every operation that

QTP does over your application. And of course you can add an Environment or Global

Parameter that skip the highlight, when running the script at night.

Overriding

```
RegisterUserFunc "Image", "Click", "ControlsImageClick", True
RegisterUserFunc "WebButton", "Click", "WebButtonClick", False
RegisterUserFunc "WebTable", "Blink", "fBlink", False
RegisterUserFunc "WebElement", "Click", "WebElementClick", False
RegisterUserFunc "WebEdit", "Set", "WebEditSet", False
```

```
Public Function WebEditSet( ByRef sender, ByVal value )

    ' ** Validation already done by ControlsBlink, no need to re-validate
    Call fBlink( sender, 2 )
    If Reporter.RunStatus = micFail Then Exit Function
    sender.Set value

End Function
```

When calling to Browser("B").Page("P").WebEdit("WE").Set "sample"

Qtp RegisterUserFunc overrides the Set method of QTP, instead WebEditSet is called.

Calling to fBlink from WebEditSet will check that, the object Exist, otherwise I will get an error message in the **Reporter**.

If the object **Exist** then a Blink process will occur during run-time and immediately the original QTP function is called from inside **WebEditSet**

API Declarations

```
Extern.Declare micHwnd, "GetDesktopWindow", "user32", "GetDesktopWindow"
Extern.Declare micULong, "GetWindowDC", "user32", "GetWindowDC", micHwnd
Extern.Declare micInteger, "ReleaseDC", "user32", "ReleaseDC", micHwnd, micULong
Extern.Declare micULong, "CreatePen", "gdi32", "CreatePen", _
    micInteger, micInteger, micDword
```

```

Extern.Declare micInteger, "SetROP2", "gdi32", "SetROP2", micULong, micInteger
Extern.Declare micULong, "SelectObject", "gdi32", "SelectObject", micULong, micULong
Extern.Declare micULong, "DeleteObject", "gdi32", "DeleteObject", micULong
Extern.Declare micULong, "GetStockObject", "gdi32", "GetStockObject", micInteger
Extern.Declare micULong, "Rectangle", "gdi32", _
    "Rectangle", micULong, micInteger, micInteger, micInteger, micInteger

```

The fBlink Function

```

' *****
' ** fBlink
' ** Description : Highlights a control specific number of times "Times"
' ** All the functions are Windows API - Extern must be declare.
' *****
Public Function fblink( ByRef sender, ByVal times )
    Const PS_SOLID = 0 : Const PS_INSIDEFRAME = 6 : Const R2_NOT = 6
    Const NULL_BRUSH = 5 : Const PEN_WIDTH = 2 :
    Dim hDC, hPen
    Dim nX, nY, nH, nW, i

    If Not sender.Exist( 1 ) Then
        Reporter.ReportEvent micFail, "fblink", "Object was not found."
        Exit Function
    End If
    ' ** Retrieve The sender information...
    With sender
        nX = .GetROProperty( "abs_x" )
        nY = .GetROProperty( "abs_y" )
        nW = .GetROProperty( "width" )
        nH = .GetROProperty( "height" )
    End With
    ' ** Get the Desktop DC
    hDC = Extern.GetWindowDC( Extern.GetDesktopWindow() )
    ' ** Create a three pixel wide pen
    hPen = Extern.CreatePen( PS_INSIDEFRAME, PEN_WIDTH, RGB( 255, 0, 0 ) )
    Extern.SetROP2 hDC, R2_NOT
    Extern.SelectObject hDC, hPen
    ' ** Use an empty fill
    Extern.SelectObject hDC, Extern.GetStockObject( NULL_BRUSH )
    ' ** Do the highlight
    For i = 0 to times * 2 + 1
        Extern.Rectangle hDC, nX, nY, nX + nW, nY + nH
        Wait 0, 50
    Next
    ' ** Release Resources '
    Extern.ReleaseDC Extern.GetDesktopWindow, hDC
    Extern.DeleteObject hPen
End Function

```

Appendix 13.A

Declare Data Types

Constant	Value	Description
micVoid	0	void (RetType only)
micInteger	2	int
micLong	3	long
micFloat	4	float
micDouble	5	double
micString	8	CHAR*
micDispatch	9	IDispatch*
micWideString	18	WChar*
micChar	19	char
micUnknown	20	IUnknown
micHwnd	21	HWND
micVPtr	22	void*
micShort	23	short
micWord	24	WORD
micDWord	25	DWORD
micByte	26	BYTE
micWParam	27	WPARAM
micLParam	28	LPARAM
micLResult	29	LRESULT
micByRef	h&4000	out
micUnsigned	h&8000	unsigned
micUChar	micChar + micUnsigned	unsigned char
micULong	micLong + micUnsigned	unsigned long
micUShort	micShort	unsigned short
micUInteger	micInteger + micUnsigned	unsigned int (UINT)

Table 1 – Declare Data Type Constants