Table of Contents

Introduction

Welcome to the online help for WebLib, Potomac Software's toolkit for creating World Wide Web applications using desktop development tools. WebLib consists of three components:

The browser API controls either the **Netscape (TM)** or **Enhanced Mosaic (TM)** Web browsers.

The toolbar API creates and attachs an application-controlled toolbar to the Web browser.

The HTML parsing API extracts information from Web documents.

Support for calling each of these APIs from C, C++ (MSVC, Borland and others), Visual Basic, PowerBuilder, Delphi and SQLWindows is included. Any other language that can call a DLL can also use the APIs.

Prerequisites

Besides having a not-too-archaic PC (386 processor or better with at least 4 megabytes of RAM), WebLib requires Microsoft Windows 3.1, a development environment of some sort (e.g., Visual Basic or C++) and most importantly, a **Netscape** or **Enhanced Mosaic** Web browser. Because WebLib acts as a wrapper around the DDE API provided by these two browsers, the version of the browser is important. WebLib works with **Netscape versions 1.1 and later** (it will not work with Netscape 1.0). While one can use version 2.0 of the Enhanced Mosaic browser from Spyglass, Inc., the DDE API in this version is limited and not particularly robust. A much better version for use with WebLib is **Enhanced Mosaic version 2.1**.

Licensing and Distribution

As one might expect, any use of WebLib is subject to and must conform with the terms of the software license agreement that comes with this package. To review the software license agreement (and we urge you to), click here.

Distribution of WebLib files is also subject to the terms of the software license agreement. Under the license agreement for the evaluation copy of WebLib, no files may be distributed whatsoever. If you buy a license for WebLib, you generally may only distribute those files needed at run-time by your applications, but you do not have to pay royalties for each copy you distribute.

Technical Support

Because the evaluation copy of WebLib is a free, beta-test version, only e-mail support is provided. Send a complete description of your question or problem to *weblib@potsoft.com*. In your message, include the Microsoft Windows version, Web browser type and version, development environment and version and any log output (see the section on Error Handling). We will try to provide as much feedback as possible but cannot guarantee a response to every message we receive.

Before you contact us with a problem or question, please check out the trouble shooting section of this document. We may already have an answer to your question!

Other Contacts

To purchase a copy of WebLib, call (301) 216-0604. For other information or queries not related to support, send e-mail to *info@potsoft.com* or call this same number. Our fax number is (301) 216-0605. You may also want to check Potomac Software's Web site from time to time to find out what's new. The URL is *http://potsoft.inter.net/potsoft/*.

Directories and Files

The directory structure used to store the WebLib files appears as follows (this diagram assumes the software was installed in the **WEBLIB** directory):

The **BIN** directory contains executable files (dynamic link libraries and programs) and help files. The MS-DOS PATH environment variable should include the BIN directory so the executable files can be found when an application that uses WebLib is run (note that during installation, all VBXs are also copied to the Windows system directory).

HTMLPARS.DLL	Dynamic link library containing HTML parsing API
HTMLPASW.APL	SQLWindows library containing declarations for HTML parsing API
HTMLPAVB.VBX	Dynamic link library containing Visual Basic HTML parsing control
WEBLIB.DLL	Dynamic link library containing browser and toolbar APIs
WEBLIB.HLP	Online help file for browser, toolbar and HTML parsing APIs
WEBLIBEX.DLL	Dynamic link library containing WebLib API extensions
WEBLIBSW.APL	SQLWindows library containing declarations for browser and toolbar APIs
WEBLIBVB.VBX	Dynamic link library containing Visual Basic browser and toolbar controls
WLSERVER.EXE	Helper application (used internally)

The **LIB** directory contains import and code libraries that allow C and C++ applications to link with the WebLib DLLs.

HTMLBCDD.LIB	Borland C++ HTML parsing class library for dynamically linked DLLs
HTMLBCDE.LIB	Borland C++ HTML parsing class library for dynamically linked EXEs
HTMLBCSD.LIB	Borland C++ HTML parsing class library for statically linked DLLs
HTMLBCSE.LIB	Borland C++ HTML parsing class library for statically linked EXEs
HTMLPARS.LIB	Import library for HTML parsing API (useful with C or C++)
HTMLPVCD.LIB	MSVC++ HTML parsing class library for AFX DLL version of MFC
HTMLPVCS.LIB	MSVC++ HTML parsing class library for static version of MFC
WEBLBCDD.LIB	Borland C++ browser/toolbar class library for dynamically linked DLLs
WEBLBCDE.LIB	Borland C++ browser/toolbar class library for dynamically linked EXEs
WEBLBCSD.LIB	Borland C++ browser/toolbar class library for statically linked DLLs
WEBLBCSE.LIB	Borland C++ browser/toolbar class library for statically linked EXEs
WEBLIB.LIB	Import library for browser and toolbar APIs (useful with C or C++)
WEBLIBEX.LIB	Import library for WebLib extension APIs (useful with C or C++)
WEBLIVCD.LIB	MSVC++ browser/toolbar class library for AFX DLL version of MFC
WEBLIVCS.LIB	MSVC++ browser/toolbar class library for static version of MFC

The **INCLUDE** directory contains text files needed by the various development environments supported by WebLib. Usually, one or more of these files must be included in your application programs.

HTMLPABC.H	Borland C++ include file for HTML parsing classes
HTMLPABC.INL	Borland C++ inline methods for HTML parsing classes
HTMLPAPB.CON	PowerBuilder constant declarations for HTML parsing API
HTMLPAPB.FUN	PowerBuilder function declarations for HTML parsing API
HTMLPARS.H	C constants and function prototypes for HTML parsing API
HTMLPASW.APT	SQLWindows text version of declarations for HTML parsing API
HTMLPAVB.H	C prototypes for action functions used with HTML parsing VBX
HTMLPAVB.TXT	Visual Basic action function declarations for HTML parsing VBX
HTMLPAVC.H	MSVC++ include file for HTML parsing classes
HTMLPAVC.INL	MSVC++ inline methods for HTML parsing classes

WEBLIB.H	C constants and function prototypes for browser and toolbar APIs
WEBLIBBC.H	Borland C++ include file for browser and toolbar classes
WEBLIBBC.INL	Borland C++ inline methods for browser and toolbar classes
WEBLIBEX.H	C function prototypes for WebLib extension functions
WEBLIBPB.FUN	PowerBuilder function declarations for browser and toolbar APIs
WEBLIBPB.CON	PowerBuilder constant declarations for browser and toolbar APIs
WEBLIBSW.APT	SQLWindows text version of declarations for browser and toolbar APIs
WEBLIBVB.H	C prototypes for action functions used with browser and toolbar VBXs
WEBLIBVB.TXT	Visual Basic action function declarations for browser and toolbar VBXs
WEBLIBVC.H	MSVC++ include file for browser and toolbar classes
WEBLIBVC.INL	MSVC++ inline methods for browser and toolbar classes

The **DOC** directory contains any documents included with WebLib.

README.TXT	Last minute release notes
LIC.TXT	License agreement

The **SAMPLES** directory contains sample code.

C	MSVC application that illustrates the use of every API via the C interface to the DLLs. This program runs stand-alone and may be used to test the WebLib APIs.
VB	Visual Basic sample app that employs the WebLib VBX controls. Requires Visual Basic 3.0.
DELPHI	Borland Delphi example program. Requires Delphi programming environment.

Troubleshooting

Here is a list of frequently asked questions (and frequently correct responses):

Q1. Why does my application fail or crash as soon as it makes a call to WebLib?

A1. Most likely, you have forgotten to call the Startup() API. When your application initializes, it must call WLStartup (for non-VBX applications) or weblibStartup (for VBX-based applications) before calling any other WebLib function. Upon termination, the application should call WLCleanup() or weblibCleanup().

Actually, Startup() is only required for the browser and toolbar APIs; you may call any of the HTML parsing APIs without calling Startup() first.

Q2. ConnectBrowser() does not start my Web browser when I pass it the WL_STARTBROWSER flag, nor does it seem to detect that the browser is running if I start the browser manually. What is going on here?

A2. One possibility might be that the **WEBLIB.INI** is not configured correctly. If you are using one of the many flavors of the Enhanced Mosaic browsers, this is more likely than with the Netscape browser.

In the [WEBLIB] section of the **WEBLIB.INI** file, the *Browser* entry should name another section in the .INI file that contains the entries *Type*, *Module*, *Program* and *DdeName*. *Type* must be 'NETSCAPE' or 'EMOSAIC' (without the quotes). The *Module* entry could be the problem. While this is typically 'NETSCAPE' for Netscape and 'EMOSAIC' for Enhanced Mosaic, the module name of your browser may be different. Under Windows 3.1, the best way to determine the correct module name is with HeapWalker: run your Web browser and then check the module names displayed by HeapWalker for a likely candidate (unfortunately, you cannot use the 16-bit version of EXEHDR against a Win32s executable such as the Netscape and Enhanced Mosaic browser). The *Program* entry should be the full path of your Web browser executable and *DdeName* should be either 'NETSCAPE' or 'MOSAIC'.

Q3. I set the AutoCreate property in the WLToolbar control to True and created some buttons at design time, but when the Visual Basic form containing the control is loaded, the toolbar is not created. What's up?

A3. If you have called weblibStartup() from the **Load event** of the form that contains the WLToolbar VBX control (or have forgotten to call weblibStartup() altogether), then setting the AutoCreate property to True will not work. This is because the toolbar is created when all controls on the form have finished loading, which happens before the **Load event** is sent to the form. You must either call weblibStartup() from **Sub Main** or set AutoCreate to False and create the toolbar yourself at run-time.

General Overview

This section presents an overview of the WebLib browser, toolbar and HTML parsing APIs and uses a pseudocode API as common ground for explaining how WebLib works. The actual APIs for C, C++ and the VBX controls are listed in the reference section.

Files

Each programming language and environment supported by WebLib requires that certain WebLib files be included in the application under development. Which files are needed and how this is done varies depending on the language and on the WebLib functionality required by the application. Generally, there are two files (or two sets of files) for each programming environment: one for the browser and toolbar APIs and one for the HTML parsing APIs. The actual file names and instructions on how to include these files into a particular programming environment are included in the overviews on C , C++ , Visual Basic , PowerBuilder , Delphi and SQLWindows .

Objects and Handles

Almost all of WebLib's functionality is supplied by three objects: the browser object, the toolbar object and the HTML parser object. Not suprisingly, there is a direct correlation between each of the three objects and each of the three APIs. Each object provides a context for the API to work in; as such, each API usually requires a handle to a valid object. For each type of object, there is an API call to create an object, returning its handle for future reference and an API to delete an object, invalidating the handle. Thus, a typical chronological sequence of API calls is:

```
// create object so we'll have it when we need it

HANDLE handle = ApiCreateObject();

// do something else... now use WebLib object and API

ApiDoThis(handle,...);
ApiDoThat(handle,...);

// do other stuff... now it's time to quit

ApiDeleteObject(handle);
```

Please note that the calls here are for illustration only.

Events

Event notification occurs in WebLib via Windows messages: when an application calls an API that generates an event, it passes the handle of a window and a message identifier to the API. To notify the application of the event when it occurs, WebLib sends the message to the window, where the application can trap it. The wParam and lParam message parameters typically contain information associated with the event. This information can be accessed by calling additional WebLib APIs.

Error Handling

Each of the WebLib APIs returns a value that may be checked to determine if the call was successful or not. In many cases, this is boolean value, where (as you would expect) TRUE indicates success. In other cases, a discrete value that indicates failure is returned if the API call is not successful (e.g., a NULL pointer is returned).

In addition, WebLib logs any internal or otherwise unexpected errors it encounters in a log file. The location and name of this log file is determined by the *File* entry in the [LOG] section of **WEBLIB.INI**.

WebLib Initialization

Before making any other calls to the browser or toolbar APIs, Startup() must be called. This function should be invoked once and only once from an application, typically when the application starts. No browser or toolbar API calls can be made unless Startup() is successfully executed first (note that the HTML parsing API is independent of Startup() and may be called at any time). Forgetting to call Startup() is a common WebLib programming mistake, so beware.

When an application terminates, it should call Cleanup() to shutdown the WebLib DLL. Like Startup(), Cleanup() should be invoked once and only once from an application. No other browser or toolbar API calls can be made once Cleanup() has been successfully executed. If an application forgets to call Cleanup() upon termination, WebLib attempts to detect that the application has died; if successful, it automatically cleans up after the expired task.

While most event notifications are particular to a specific object and API (e.g., browser), the SetDefaultNotify() API lets you establish a notification window and message that spans APIs. In subsequent calls that require a notification window and message, you can use the constant WL_DEFAULTNOTIFY to specify the window and message set in SetDefaultNotify(). More importantly, calling SetDefaultNotify() lets you trap the BrowserStart and BrowserExit events. This lets you know if the Web browser is started or terminated while your application is running.

Browser API

The browser API provided by WebLib is a wrapper around the DDE APIs provided by the **Netscape** and **Enhanced Mosaic** Web browsers. WebLib's browser API has the advantage of being much easier to use than the DDE APIs, while still being based on a standard, vendor-supported and documented browser interface. In addition, even though the DDE APIs in the **Netscape** and **Enhanced Mosaic** browsers are similar, there are several significant differences. The WebLib browser API hides these differences so that is possible to write a single set of source code that will work with both browsers.

The first browser API call an application should make is ConnectBrowser(). This creates a browser object and returns its handle for use in subsequent calls. This API also lets you start the Web browser if it is not already running. The complementary API to ConnectBrowser() is DisconnectBrowser(). This terminates the logical connection between the application and the Web browser and frees the browser object. If specified, DisconnectBrowser() will also attempt to terminate the Web browser program.

Notification Methods

The ConnectBrowser() API also lets you set the notification method used by WebLib to send events to an application window. This is either WL_POSTMESSAGE (notify via the **PostMessage** SDK call) or WL_SENDMESSAGE (notify via the **SendMessage** SDK call). There are advantages and disadvantages to both methods, but we strongly recommend that WL_POSTMESSAGE (the default) be used whenever possible.

The biggest drawback of WL_SENDMESSAGE is that it is dispatched from the DDE callback that makes up one end of the DDE conversation between WebLib and the Web browser. As such, the application must return control to **SendMessage** quickly to avoid a timeout in the DDE conversation. In addition, using WL_SENDMESSAGE may require the app to use the **ReplyMessage** SDK call to avoid deadlock, such as before displaying a message box or dialog. Note that popping up a message box or dialog box upon receipt of a browser event sent via **SendMessage** has its own problems, namely the response time problem mentioned before: you never know how long it will take the end-user to dismiss the message box or dialog.

The primary disadvantage of WL_POSTMESSAGE is the possibility of filling up the application's message queue. During testing we have not seen this happen; also, one may set the size of the queue with the **SetMessageQueue** SDK call (or in SYSTEM.INI).

The browser notification method can be changed using SetNotifyMethod().

Window Manipulation

The browser API includes a set of calls to manipulate the Web browser's windows. The ListWindows() API can be called successively to enumerate all of the open browser windows. GetWindowInfo() returns the URL and title of a specific browser window. A browser window may be activated (brought to the top) with ActivateWindow() and may be closed with CloseWindow(). To change the size and position of a window, call SetWindowPos(). The ShowWindow() API lets you mimimize, maximize or restore a browser window.

An application can receive notification when a browser window is sized, moved, minimized, maximized or closed by calling the RegisterWindowChange() API.

Loading URLs

The OpenURL() API loads the document specified by a URL and displays it. The SaveURL() API is similar, expect that it saves the document in a local file after loading it. Both of these functions return a transaction ID that may be used to monitor and control the loading of a particular URL. A transaction that is in progress may be canceled by passing the transaction ID to Cancel(). The transaction ID can also be used to determine which browser window the URL was displayed in by calling GetTransactionWindow().

OpenURL() and SaveURL() can both generate several events (event notification is optional; if you don't want notification, pass the constant WL_NONOTIFY instead of the handle of the window to notify). The events that may be generated by these two browser APIs are listed below in the sequence they are generated:

BeginProgress
SetProgressRange
MakingProgress (one or more)
EndProgress
Finished (or Canceled)

Of these events, only the Finished or Canceled event is guaranteed to be generated. While the progress events are usually generated, certain cases exist when they are not (e.g., loading a URL that is already displayed under **Enhanced Mosaic**). To be safe, use only the Finished or Canceled event to determine the state of a transaction; the progress events should only be used to report feedback to the user (which is the intent of these events, anyhow).

Posting Form Data

Two browser APIs let you send data to a URL using HTTP's POST method: PostFormData() and SaveFormData(). Once the data is posted to the URL, the result sent back from the server is displayed in the specified browser window or saved to a file.

These APIs require you to specify the MIME type of the data being posted to the URLs. Typically, the data is url-encoded (i.e., ASCII text with spaces and unprintable characters replaced with a plus sign or other printable character) and consists of one or more 'field=value' tokens so as to mimick the data format used by HTML forms. To assist in creation of this type of form data, WebLib provides the AppendFormData(), GetFormDataLength() and AccessFormData() APIs. These functions make it easy to build a variable-length buffer of url-encoded, 'field=value' strings.

PostFormData() and SaveFormData() generate the same events as OpenURL() and SaveURL(), i.e., the progress events followed by Finished or Canceled. The same caveats about counting on particular events always being generated also apply.

Monitoring, Overriding and Extending the Web browser

Three sets of browser APIs let you monitor and takeover certain functions of the Web browser software.

The RegisterProtocol() API directs the Web browser to let the application handle certain protocols. This can be a standard, built-in protocol like 'http' or 'ftp' or it may be a user-defined protocol that only you know to handle. When the end-user tries to load a URL that employs a protocol that has been overridden via RegisterProtocol(), the Web browser (via WebLib) sends the OpenURL event. The UnregisterProtocol() API informs the Web browser that the application no longer wants to handle a given protocol.

An application can handle the viewing of documents of a particular MIME type by calling the RegisterViewer() API. The flags passed to this function determine how the Web browser interacts with the application to view the document and must be well understood if this feature is to work properly. Meaningful flag values include:

WL_SHELLEXECUTE	The Web browser calls the **ShellExecute**() SDK function to invoke an program to view the document. Because the filename extension must exist in the registration database to launch a viewer, this often fails. For further information, see the SDK documentation about the **Shell library**.
WL_VIEWDOCFILE	Via WebLib, the Web browser sends a ViewDocFile event to the application. The application is responsible for displaying the document, which is stored in a file whose name is included with the event.
WL_QUERYVIEWER	This flag should be combined with either of the above flags. It causes the Web browser (via WebLib) to send a QueryViewer event to the application first. This gives the application a chance to specify the name of the file that the document will be saved in. The application does this with the SetFileName() API. The filename is then passed along to **ShellExecute**() or included with the ViewDocFile event.

To inform the Web browser that an application no longer wishes to handle the viewing of documents of a certain MIME type, call the UnregisterViewer() API.

Passive monitoring of the Web browser is possible via the RegisterURLEcho() API. This function causes the browser software to report the loading of URLs to the application. Thus, as the end-user 'surfs the Net', an application can track where the user has been. The application receives a URLEcho event for each URL that is loaded (actually, this behavior varies slightly between **Netscape** and **Enhanced Mosaic**; Enhanced Mosaic only reports new URLs). The UnregisterURLEcho() API turns off the echoing of URLs.

As mentioned above, RegisterWindowChange() allows an application to monitor changes in a browser window's geometry and state.

Miscellaneous Browser APIs

The ShowFile() API displays a local file in a browser window. It generates the same events as OpenURL(). To get the version of the DDE interface offered by the Web browser, call GetVersion(). To

combine a relative URL with an absolute URL, use the ParseAnchor() API.

An API that is only available with the **Netscape** browser is QueryURLFile(). Given the name of a local file that was loaded by the Web browser, QueryURLFile() returns the URL the file was loaded from or is associated with. For **Enhanced Mosaic**, this API is a no-op.

Toolbar API

The WebLib toolbar API allows an application to create a customized toolbar and attach it to the Web browser. When the end-user clicks one of the toolbar buttons, the application receives an event.

How the Toolbar Works

Here is an example toolbar:

When the end-user clicks the right mouse button in the text area of the toolbar, a popup menu appears. This menu lets the user do several things:

- Attach the toolbar to any side of the Web browser (but only to the outside of the main browser window).
- Minimize the toolbar.
- Select another toolbar and make it active. WebLib allows multiple toolbars to be created and overlaid in a single toolbar frame.
- Toggle the Open On Browser Maximize feature. When checked, this option causes the toolbar to be restored from a minimized state when the browser is maximized. This allows WebLib to reserve enough space on the display for the toolbar (after the Web browser is maximized, the toolbar may not be attached to a different side). If this option is not checked, the toolbar will overlay part of the browser window if it is restored from a minimized state and the Web browser window is maximized at that point.

The Web browser and toolbar act in concert whenever possible. When the browser is activated, the toolbar comes to the foreground with it; when the size or position of the browser window changes, the toolbar stretches or shrinks to fit the new size or moves to the new position. If the Web browser is minimized, the toolbar hides itself; when the browser window is restored, so is the toolbar. The toolbar also hides itself if the browser is closed; when the Web browser is started again, the toolbar re-appears.

Three types of text strings may be displayed on the toolbar:

- Each button may have a description associated with it that appears when the user places the mouse cursor over the button.
- The toolbar itself may have a description that appears in the text area of the toolbar
- Each toolbar has a title that appears in the popup menu.

WebLib allows you to get and set these text strings, and to change the color and font used to display the strings.

From the programmer's perspective, adding a button to the toolbar should be viewed as placing it into an imaginary array of buttons that starts at the left side of the toolbar (position zero) and stretches to the right. The rightmost position in the array that contains a button marks the beginning of text area of the

toolbar. The array positions between position zero and the rightmost occupied position may or may not contain buttons. Adding a button to a position that already contains a button destroys the existing button. Buttons may be hidden or disabled.

Programming the Toolbar

CreateToolbar() is used to create a toolbar. This API returns a handle for use in subsequent toolbar API calls. It also specifies which window WebLib should send the ButtonClick event to. The DeleteToolbar() destroys a toolbar and invalidates the handle associated with the toolbar.

When a toolbar is created, it becomes the active toolbar. The active toolbar may be changed via the SetActiveToolbar() API. An application can determine if a particular toolbar is active by calling IsToolbarActive().

Buttons are placed on the toolbar with the AddToolbarButton() API. This function requires a unique button ID that acts as an identifier, a button position (zero is the leftmost button), a string that describes the function of the button and up to four bitmaps. Each bitmap **must be 24 x 24 pixels** in size. A bitmap may be specified for each of the following button states: normal (up), selected (down), input focus (usually a dashed line is drawn around the edge of the button) and disabled. Only a bitmap for the normal state is required. AddToolbarButton() requires that the bitmaps be specified as resources in a program or DLL. The WebLib extension APIs **AddToolbarButtonByHandle**() and **AddToolbarButtonByFile**() provide additional flexibility: the former adds a toolbar button using bitmap handles while the latter loads the bitmaps for the button from files.

To delete a toolbar button, call RemoveToolbarButton().

A button can be hidden or shown using the ShowToolbarButton() API and it's visibility can be checked with IsToolbarButtonVisible(). Similarly, a button can be disabled or enabled using the EnableToolbarButton() API; whether or not the button is enabled can be checked with IsToolbarButtonEnabled().

To get and set the descriptive text associated with the toolbar or a particular button, call GetToolbarText() and SetToolbarText(), respectively. The font used to display such text may also be obtained and set via GetToolbarFont() and SetToolbarFont(). When setting a font, the application must create a font and pass its handle to SetToolbarFont(); be careful not to delete this font until the toolbar has been destroyed.

Toolbar colors may be changed by calling SetToolbarBkgnd() to set the background color and SetToolbarTextColor() to set the colors for displaying button and toolbar text (each of these may be set to a different color).

HTML Parsing API

The HTML parsing APIs fall into three catgeroies: functions that break down an HTML document into individual elements, functions that access the individual elements and functions that find particular elements.

Parsing the HTML Document

The first category of APIs includes HtmlParseFile(), HtmlParseBuf() and HtmlEndParse(). HtmlParseFile() parse an HTML document stored in a file, while HtmlParseBuf() parses an HTML document stored in a memory buffer. Both of these functions return a parse handle that must be passed to subsequent API calls.The HtmlEndParse() API frees the memory associated with the parse handle, invalidating it.

HtmlParseFile() and HtmlParseBuf() are the real workhorses of the HTML parsing API. These functions use a recursive algorithm to create a tree of basic HTML elements like tags and text strings. This work is CPU and memory intensive and requires a reasonably large stack. HTML documents containing many

thousands of elements can take several seconds to parse.

The parse tree created by HtmlParseFile() and HtmlParseBuf() has a root element that serves as a starting point for the rest of the tree. Each level of the tree corresponds to an HTML container tag (e.g., <BODY> and </BODY>). Thus, the only elements in the tree with children are container tag elements and the root element. Other elements like empty tags and text are strung along at the same level as siblings in a linked list. Each of these other elements has a parent, either a container tag or the root element.

Tags that simply change the appearance of text (e.g., <B> and </B>) are treated differently from other tags. When a tag of this type is recognized by the parser, a bit in the text attribute bitmask is set or cleared depending upon whether that tag is an opening or closing tag. This text attribute bitmask is then associated with the text that occurs between the opening and closing tag. Thus, a new text element is created each time a text attribute tag is encountered (i.e., when the appearance of the text changes). Every text element has a text attribute bitmask associated with it; if no text attribute tags enclose the text, the bitmask is zero.

Every element in a parse tree is one of five types: WL_TAG (includes container tags, empty tags and text attribute tags), WL_TEXT (text string with a text attribute bitmask), WL_SPECIALCHAR (things like < and >), WL_COMMENT and WL_ROOT (root element).

Tag elements contain tag attributes (not to be confused with text attributes). These are tokens embedded in the tag that usually look like variable assignments (e.g., ALIGN=TOP). The HTML parser stores the left side of such assignments as the tag attribute name and the right side as the tag attribute value. Standalone tags are also supported; these are single tokens that are not part of an assignment. In fact, every tag element contains at least one standalone tag attribute: the tag name itself (e.g., BODY).

Because the HTML elements between opening and closing container tags are stored as children of the opening container tag, the default for HtmlParseFile() and HtmlParseBuf() is not to store the closing tag. Likewise, the default for HtmlParseFile() and HtmlParseBuf() is not to store text attribute tags such as <B> and </B> since the text attribute bitmask makes these tags superfluous. Pass the constants WL_KEEPCLOSINGTAG and WL_KEEPATTRIBUTETAG to HtmlParseFile() and HtmlParseBuf() to retain closing container tags and text attribute tags.

The parse tree described above is best explained using an example HTML document and a diagram of the tree produced by parsing the example document.

Accessing Elements in the Parse Tree

Functions that access the individual elements in the parse tree are the second category of HTML parsing APIs. Elements in the tree are referenced by an element handle; most APIs accept an element handle that specifies where in the parse tree processing should begin. The constant WL_ROOTELEMENT may be used to access the root of the parse tree.

Every element in the parse tree may be enumerated by calling the EnumerateParseTree() API. This function sends a message to a window for each element in the tree (the message identifier and the window are specified to the API). The *wParam* message parameter contains the element type and the *lParam* message parameter contains the element handle. The application must return TRUE in response to the message to continue the enumeration.

HtmlGetChild() get the first element in the next level of the tree, while HtmlGetParent() returns the element at the previous level. The HtmlGetSibling() API returns the next, previous, first or last element at the same level (and under the same parent).

Given an element handle, you can determine the element' s type via HtmlGetElementType(). The text associated with the element is returned by HtmlGetElementText().

For text elements, the text attribute bitmask may be obtained by calling HtmlGetTextAttr(). The handle of a tag element may be passed to HtmlGetTagName() and HtmlGetTagType() to determine -- yes, you guessed it -- the tag name (e.g., "BODY") and type (e.g., HTML_BODY).

The attributes in a tag may be enumerated by making successive calls to HtmlGetTagAttr(). To get the value of a particular attribute in a tag, use the HtmlExtractTagAttr() API.

Finding HTML Elements

The third and last category of HTML parsing API searches the parse tree (or part of it) for elements that are of a particular type or contain a particular value.

HtmlFindText() locates the next text element in the tree that matches a specified value. Similarly, the HtmlFindSpecial() and HtmlFindComment() APIs find the next special character or comment element that matches a given value.

The HtmlFindTagType() and HtmlFindTagName() APIs search for the next element in the tree that matches the specified tag type or name, respectively. The next tag element containing an attribute with a certain value can be found using the HtmlFindTagAttr() API.

For each of the 'find' APIs described above, a corresponding 'find and enumerate' API exists. Instead of finding just the next occurrence of an element, the enumeration APIs send a message to a window for each occurrence of the element that is found. This is similar to HtmlEnumParseTree(), except that only a subset of the elements in the tree are enumerated to the application. The 'find and enumerate' APIs are HtmlEnumFindText(), HtmlEnumFindSpecial(), HtmlEnumFindComment() HtmlEnumFindTagType(), HtmlEnumFindTagName() and HtmlEnumFindTagAttr().

Visual Basic Overview

WebLib supports Visual Basic with three VBX controls: WLBrowser, WLToolbar and WLParser. Although we recommend using Visual Basic 3.0, the controls have been written to support Visual Basic 1.0.

To include WebLib into your VB project, do the following:

- Pick Add File from the File menu and select the file **WEBLIBVB.VBX** from the Windows system directory to include the browser and toolbar VBXs. Add the file **HTMLPAVB.VBX** in similar fashion to include the HTML parsing VBX.

- Choose the Load Text option from the File menu and select the file **WEBLIBVB.TXT** if you plan to use the browser or toolbar VBX and **HTMLPAVB.TXT** if you plan to use the HTML parsing VBX. These files include the external declarations of the action functions that are used to call an API from a VBX.

Properties

Most of the properties in the WebLib VBXs provide only run-time support. The exception is the WLToolbar VBX, which includes a large number of properties that can be set at design-time. Two very important run-time propeties are included in each control: Action and Result. These properties are used to invoke an API and are explained below.

Events

Each VBX generates events to notify the application of some phenomenon. One or more parameters are included with each type of event.

Under Visual Basic, care must be taken to avoid 'eating' events that have been queued via the **PostMessage** SDK call, awaiting dispatch after the current event handler returns. VB functions like **MsgBox**() do just this, causing a loss of some events. One way to get around this is to use WL_SENDMESSAGE as the notification method instead of WL_POSTMESSAGE, but this has it owns problems, as explained in the General Overview. Our advice is to avoid calling **MsgBox**() and its ilk from an event handler.

If an API generates events, event generation may be disabled for the API by setting the GenerateEvents property to False. This property is automatically reset to True after each API call, so GenerateEvents must be set to False just before calling the API to disable event generation.

Initialization

Applications must call weblibStartup() and weblibCleanup() before calling any other VBX-based APIs. The best place to do this is **Sub Main**; however, weblibStartup() and weblibCleanup() cannot both be called from **Sub Main** unless the forms created by the application are modal. For apps that use MDI windows or other non-modal features, only weblibStartup() can be called from **Sub Main**. Invoking weblibStartup() from the **Load form event** and weblibCleanup() from the **Unload form event** is another possibility, but only if one form needs to use the WebLib VBXs.

If multiple, non-modal forms need to use the WebLib VBXs, we recommend a couple of approaches. The first involves calling weblibStartup() from **Sub Main** and then keeping a reference count of loaded forms (i.e., add one to a global counter when a form is loaded and subtract one from the counter when a form is unloaded). When the reference count reaches zero (the last form has been closed), call weblibCleanup(). Another approach is to create an invisible form whose sole purpose is to call weblibStartup() on form load and weblibCleanup() on form unload. The application creates this form first and unloads it last.

As explained below, the AutoCreate property of the WLToolbar VBX can cause problems when weblibStartup() is called from the **Load form event**.

Note that if weblibStartup() is called from **Sub Main** and the VB app terminates without calling weblibCleanup(), WebLib usually cleans up after itself anyway (this behavior occurs only when the VB app is run stand-alone).

Calling an API

To call a browser, toolbar or HTML parsing API using a VBX, simply set the Action property to the return vaue of the appropriate action function. Then check the Result property to determine of the API call was successful or not. If the call was successful (the Result property is True), additional properties that may have been set by the API can be accessed. If the Result property is False, meaning that the API call failed, **do not access any other properties**; doing so causes VB error 390, "No defined value".

Here is an example that grabs the URL and title of window using the browser VBX:

```
WLBrowser.Action = actionGetWindowInfo(WL_ACTIVEWINDOW)
If (WLBrowser.Result) Then
      MsgBox "URL = " & WLBrowser.URL & " Title = " & WLBrowser.Title
Else
      MsgBox "actionGetWindowInfo failed"
End If
```

Toolbar Design-time Properties

The following WLToolbar properties may be set at design-time:

AutoCreate	Button	ButtonID	ButtonPic
ButtonPicDisabled	ButtonPicFocus	ButtonPicSel	ButtonText
ColorBkgnd	ColorButtonText	ColorToolbarText	FontButton
FontToolbar	TextMenu	TextToolbar	

While there is nothing special about most of these properties, the Button, ButtonID, ButtonPic, ButtonPicDisabled, ButtonPicFocus, ButtonPicSel and ButtonText properties are different. To add a button to the toolbar at design-time, first select one of the twelve button positions specified by the Button property. Then provide values for the ButtonID, ButtonPic, ButtonPicDisabled, ButtonPicFocus, ButtonPicSel and ButtonText properties (these last four properties are optional; setting them creates a nicer-looking toolbar).

The ButtonID property is key when adding toolbar buttons at design-time. This should be a unique, non-zero ID for the button. If ButtonID is zero, then no toolbar button will be added at the position specified by the Button property. This is how you clear a button at a given position.

The other design-time properties may be reset so that they will be ignored when the toolbar is created by providing a NULL value. Generally, this is accomplished by selecting the property in the Visual Basic Properties window and press **F2** followed by the **Delete** key. For text and font properties, this results in an empty string; for bitmaps, the value (None) appears to indicate the property has been reset. Color properties are different; because Visual Basic does not allow the hig-order byte of the color to be set, a special **NULL color value** for the other three bytes must be used. By default, this is &H00AAAAAA&. If you need to use the color represented by &H00AAAAAA&, you can change the **NULL color** by editing the *NullColor* entry in the [VBX] section of **WEBLIB.INI** (if this entry is changed, any color properties in existing WLToolbar VBX controls that use &H00AAAAAA& as the default color also have to be changed).

When set to True, the AutoCreate property causes the toolbar to be created at the same time as the VBX, i.e., when a form containing the WLToolbar VBX control is loaded. The toolbar is deleted when the form is unloaded. This avoids having to call the functions actionCreateToolbar() and actionDeleteToolbar() at run-time. However, the toolbar will not be created even when AutoCreate is True if weblibStartup() is called from the **Load event** of the form that contains the WLToolbar VBX control. This is because the **Load event** is sent **after** the VBX is created and thus after the VBX tries to create the toolbar. For this and other reasons, it is best to place weblibStartup() in **Sub Main** whenever possible.

When a toolbar is created using the run-time function actionCreateToolbar(), it is initialized with any design-time properties that have been set to a valid (i.e., non-NULL) value.

Adding Toolbar Buttons at Run-time

Three functions exists for adding toolbar buttons at run-time: actionAddToolbarButtonByHandle(), actionAddToolbarButtonByID() and actionAddToolbarButtonByName(). The latter two functions load the bitmap as a resource from an executable or DLL and require an instance handle, which typically involves making a call to the Windows SDK.

The actionAddToolbarButtonByHandle() function is easier to use from Visual Basic. It adds a toolbar button based on a bitmap handle, which can be obtained using the **Image** property of a VB **picture box** control. Thus, one might create a series of invisible **picture box** controls, associate them with various bitmaps and create toolbar buttons by passing the **Image** properties of these controls to actionAddToolbarButtonByHandle(). If you use this approach, you must have actionAddToolbarButtonByHandle() make a copy of the bitmap or things will go very badly.

Limited Stack under Visual Basic

Given the recursive nature of the HTML parsing APIs and the fact that Visual Basic has a fixed stack of about 20K, it is possible that certain deeply nested HTML constructs could cause a stack fault. We haven't seen this happen, but watch out anyway!

PowerBuilder Overview

WebLib supports PowerBuilder via a set of wrappers around the C API. Due to the limited nature of the VBX support provided by PowerBuilder, the WebLib VBXs cannot be used.

The constants and external function declarations needed by PowerBuilder are contained in plain text files and must be copied and pasted into the PowerBuilder environment. To do this, run a application such as NOTEPAD, load the WebLib file and copy the contents of the entire file to the clipboard. The bring up the proper PowerBuilder window and paste the contents of the clipboard to the end of any declarations that may already be present.

To include WebLib constants into your PowerBuilder project, paste the constants you need into the dialog that is invoked by the Global Variables option under the Declare menu. For the browser and toolbar APIs, copy and paste the contents the file **WEBLIBPB.CON**. For the HTML parsing API, copy and paste the contents of the **HTMLPAPB.CON** file.

To include WebLib external function declarations into your PowerBuilder project, paste the declarations you need into the dialog that is invoked by the Global External Functions option under the Declare menu. Copy and paste the contents the file **WEBLIBPB.FUN** for the browser and toolbar APIs. For the HTML parsing API, copy and paste the contents of the **HTMLPAPB.FUN** file.

Initialization

WebLib must be initialized before using any of the browser or toolbar APIs. To do this, call WLStartup() from the application open event. To terminate WebLib, call WLCleanup() from the application close event.

Handling Events

Events are handled as they are in the C API, by sending Windows message to a particular window. In PowerBuilder, this involves using the **Handle**() function and the system-defined **Message** object. To obtain the handle of the window that is to receive the message, use **Handle**(). To reference message parameters and to return a value from a message handler, use the **Message** object.

The pre-defined message identifier WM_WEBLIB_NOTIFY is included for use with the browser and toolbar APIs. In addition, the message identifiers WM_WEBLIB_ENUMPARSETREE, WM_WEBLIB_ENUMFINDTEXT, WM_WEBLIB_ENUMFINDSPECIAL WM_WEBLIB_ENUMFINDCOMMENT, WM_WEBLIB_ENUMFINDTAGTYPE, WM_WEBLIB_ENUMFINDTAGNAME, WM_WEBLIB_ENUMFINDTAGATTR have been pre-defined for use with the HTML parsing API. These message identifiers have been assigned in such a way that there should not be any conflict with any standard controls or windows.

Handling of the pre-defined messages should occur through PowerBuilder's **Other** window event. Another approach is to create your own user-defined events in PowerBuilder and use these for notification messages instead of the pre-defined identifiers.

Here is an example of how to setup and receive URLEcho events:

```
      // Register app to receive URLEcho notifications in script for a menu
item

      IF NOT
WLRegisterURLEcho(h_browser,Handle(ParentWindow),WM_WEBLIB_NOTIFY)) THEN
            MessageBox("Error","WLRegisterURLEcho failed");

      // Script for Other event in ParentWindow of menu item
```

```
	ulong l_window
	string s_url
	string s_mime
	string s_ref
	string s_msg

	IF Message.Number = WM_WEBLIB_NOTIFY THEN
		CHOOSE CASE Message.WordParm
			CASE WLN_URLECHO
				l_window = WLNGetWindow(Message.LongParm)
				s_url = WLNGetURL(Message.LongParm)
				s_mime = WLNGetMIMEType(Message.LongParm)
				s_ref = WLNGetReferrer(Message.LongParm)
				s_msg = String(l_window,"[General]") + s_url + s_mime
+ s_ref
				MessageBox("Received URLECHO event!",s_msg)

			CASE WLN_XXX
				// more event handling here...

			CASE WLN_YYY
				// ...and here

		END CHOOSE
	END IF
```

The enumeration functions in the HTML parsing API require that TRUE be returned from the message handler to continue the enumeration. To do this, set **Message.ReturnValue** to the value you want to return (1 for TRUE, 0 for FALSE) and set **Message.Processed** to TRUE. Both assignments must be performed and they must be done in this order or no value will be returned:

```
	Message.ReturnValue = 1		// or 0 for FALSE
	Message.Processed = TRUE
```

Returning Strings from a API

Certain APIs such as WLGetWindowInfo() pass one or more strings back as parameters. Before calling the API, any strings that are to receive values must be filled out to their maximum expected length with the PowerBuilder **String**() function (256 is a good maximum length to use). For example:

```
	string s_url = Space(256)
	string s_title = Space(256)

	IF NOT WLGetWindowInfo(h_browser,WL_ACTIVEWINDOW,s_url,256,s_title,256)
THEN
		MessageBox("Error","WLGetWindowInfo failed");
	END IF
```

Failing to allocate enough space in a string that can be modified by an API can cause truncated values or worse.

Adding Toolbar Buttons

The WebLib extension APIs include several functions for adding a button to a toolbar that are easier to use with PowerBuilder than WLAddToobarButton(). In particular, the extension function

WLAddToobarButtonByFile() works well with PowerBuilder since it loads bitmaps from files. The bitmaps stored in the files must be DIB-compatible but not compressed (this is usually the case with files having the .BMP extension).

SQLWindows Overview

SQLWindows is supported by Welib via a set of wrappers around the C API. The VBX support provided by SQLWindows is not robust enough to allow the use of the WebLib VBXs.

Two SQLWindows 5.0 libraries are included with WebLib: **WEBLIBSW.APL** contains the contants and declarations for the browser and toolbar APIs, while **HTMLPASW.APL** contains the constants and declarations for the HTML parsing API. These should be added to the outline of your SQLWindows application under Libraries as File Include: **WEBLIBSW.APL** and File Include: **HTMLPASW.APL**.

You may need to change the File Path in the SQLWindows Preferences dialog to include the **WEBLIB\BIN** directory.

Two alternative files, **WEBLIBSW.APT** and **HTMLPASW.APT**, are included in case you cannot use the compiled libraries **WEBLIBSW.APL** and **HTMLPASW.APL**. Thes files can be found in the **WEBLIB\INCLUDE** directory. **WEBLIBSW.APT** and **HTMLPASW.APT** are skeleton SQLWindows 5.0 apps that have been saved as text files. The idea is to load these projects into SQLWindows and then copy and paste the WebLib global constants and external function declarations from these skeleton apps into your SQLWindows applications.

Initialization

Before using any of the browser or toolbar APIs, WebLib must be initialized. To do this, call WLStartup() from the **On SAM_AppStartup** message handler. To terminate WebLib, call WLCleanup() from the **On SAM_AppExit** message handler. Both of these things are done under Global Application Actions in the SQLWindows outline.

Handling Events

Events are handled as they are in the C API, by sending Windows message to a particular window. In SQLWindows, use the system-defined global variables **hWndForm**,**wParam** and **lParam**.

Pre-defined message identifiers are included as global constants in the SQLWindows libraries that come with WebLib. Use the message identifier WM_WEBLIB_NOTIFY with the browser and toolbar APIs. The message identifiers WM_WEBLIB_ENUMPARSETREE, WM_WEBLIB_ENUMFINDTEXT, WM_WEBLIB_ENUMFINDSPECIAL WM_WEBLIB_ENUMFINDCOMMENT, WM_WEBLIB_ENUMFINDTAGTYPE, WM_WEBLIB_ENUMFINDTAGNAME, WM_WEBLIB_ENUMFINDTAGATTR should be used with the HTML parsing API. These message identifiers have been assigned in such a way that there should not be any conflict with any standard controls or windows.

In the outline, message handing usually occurs in the Message Actions section for a form or window. The following is an example of how to setup and receive URLEcho events:

```
      ! ===== Register app to receive URLEcho notifications in actions for
menu item

      If NOT WLRegisterURLEcho(hBrowser,hWndForm,WM_WEBLIB_NOTIFY))
            Call SalMessageBox('WLRegisterURLEcho failed','Error',MB_Ok)

      ! ===== form variables needed by message actions

      Number: nWindow
      String: sURL
      String: sMIME
```

```
      String: sRef
      String: sMsg

      ! ===== actions for On WM_WEBLIB_NOTIFY in message action section of
outline

      Select Case wParam
            Case WLN_BEGINPROGRESS
                  Set nWindow = WLNGetWindow(lParam)
                  Set sURL = WLNGetURL(lParam)
                  Set sMIME = WLNGetMIMEType(lParam)
                  Set sRef = WLNGetReferrer(lParam)
                  Set sMsg = SalNumberToStrX(nWindow,0) || sURL || sMIME ||
sRef
                  Call SalMessageBox(smsg,'BEG',MB_Ok)

            Case WLN_XXX
                  ! ===== more event handling here...

            Case WLN_YYY
                  ! ===== ...and here
```

The enumeration functions in the HTML parsing API require that TRUE be returned from the message handler to continue the enumeration. In SQLWindows, this is simple: just use the **Return** statement to return TRUE or FALSE. For example:

```
      On WM_WEBLIB_ENUMPARSETREE
            ! ===== do something useful here...
            Return TRUE
```

Returning Strings from a API

Certain APIs such as WLGetWindowInfo() pass one or more strings back as parameters. Before calling the API, any strings that are to receive values must be filled out to their maximum expected length with the **SalStrSetBufferLength**() function (256 is a good maximum length to use). For example:

```
      ! ==== variables

      String: sURL
      String: sTitle

      ! ===== actions

      Call SalStrSetBufferLength(sURL,256)
      Call SalStrSetBufferLength(sTitle,256)

      If NOT WLGetWindowInfo(hBrowser,WL_ACTIVEWINDOW,sURL,256,sTitle,256)
            Call SalMessageBox('WLGetWindowInfo failed','Error',MB_Ok);
      END IF
```

Failing to allocate enough space in a string that can be modified by an API can cause truncated values or worse.

Adding Toolbar Buttons

The WebLib extension APIs include several functions for adding a button to a toolbar that are easier to

use with SQLWindows than WLAddToobarButton(). In particular, the extension function **WLAddToobarButtonByFile**() works well with SQLWindows since it loads bitmaps from files. The bitmaps stored in the files must be DIB-compatible but not compressed (this is usually the case with files having the .BMP extension).

C API Overview

The C API lets C and C++ applications call the WebLib browser, toolbar and HTML parsing APIs.

To use the browser and toolbar APIs in a Windows-based C program, include the file **WEBLIB.H** in your source code. To use the HTML parsing API, include the file **HTMLPARS.H**.

C applications that use WebLib make direct calls to **WEBLIB.DLL** and **HTMLPARS.DLL**. As such, the import libraries **WEBLIB.LIB** and **HTMLPARS.LIB** must be linked into an application that makes WebLib API calls. Link with **WEBLIB.LIB** if the app uses the browser or toolbar APIs and **HTMLPARS.LIB** if the app calls any of the HTML parsing APIs.

Initialization

To use any of the browser or toolbar APIs, C language applications must call WLStartup() upon startup and should call WLCleanup() upon application termination.

Events

Event notification is performed by sending a Windows message to a window whose handle was specified in an API call. Note that you can use a single message identifier for all of the browser and toolbar APIs that generate events, since the window that receives the message can check wParam to determine the type of event. This is in contrast to the HTML parsing APIs, which require a distinct message for each enumeration function. Of course, you may also pass a distinct message identifier to each event-generating browser API. Which method you use is a matter of style.

Upon receiving an event, the values bundled into the *lParam* message parameter can be extracted with the **WLNGetXXX** functions. The values bundled with an event vary based on the event type. Here is an example that handles browser and toolbar events by switching on the notification code (i.e., *wParam*):

```
      switch (Message)
      {
            case WM_WEBLIB_NOTIFY:
                  switch (wParam)
                  {
                        case WLN_BEGINPROGRESS:
                        {
                              DWORD dwTrx = WLNGetTransaction(lParam);
                              const char *pszProgress =
WLNGetProgressString(lParam);
                              // do something with dwTrx and pszProgress...
                              break;
                        }

                        case WLN_SETPROGRESSRANGE:
                        {
                              DWORD dwTrx = WLNGetTransaction(lParam);
                              DWORD dwMax = WLNGetProgressMaximum(lParam);
                              // do something with dwTrx and dwMax...
                              break;
                        }
                        ...
                  }
                  break;
```

```
        case WM_XXX:
        ...
}
```

The WebLib extension library **WEBLIBEX.DLL** contains some useful functions for adding toolbar buttons. Included in this library are **WLAddToolbarButtonByHandle**(), **WLAddToolbarButtonByID**(), **WLAddToolbarButtonByName**() and **WLAddToolbarButtonByFile**(). These functions are generally easier to use than WLAddToolbarButton(), which must accomodate the passing of bitmaps by handle, resource ID and resource name in one API.

Compiling and Linking

It is recommended that C apps using the WebLib APIs be developed using the large memory model and have a stack of at least 24K.

C++ Overview

WebLib's C++ support includes classes for MSVC++/MFC, Borland C 4.5/OWL and other C++ compilers. Although different classes are provided for each compiler and class library, these classes are very similar.

Class Overview

The WebLib APIs are supported via two types of classes: a simple API wrapper class and a slightly more complex event-handling class which is derived from the API wrapper class. In the following discussion, the MSVC++/MFC classes are used as examples; the same principles apply to the Borland C++/OWL classes.

The API wrapper classes include a method for each WebLib API they encapsulate. Their primary advantages over the C API are that the object handle (e.g., HBROWSER) is stored in the class instance and that the constructor and destructor can allocate and free objects automatically. Event notification takes place as it does in the C API, via a window handle that is passed to a method.

The event-handling classes take things one step further than the API wrapper classes. These classes handle events via virtual methods. Instead of passing the handle of an unrelated window that is to be notified when an event occurs, instances of the event-handling classes notify themselves of events. Given this, the event-handling classes are designed for derivation: to be notified of events, one should derive a new class from one of the event handling classes and override the appropriate virtual methods.

Another difference between the API wrapper classes and the event handling classes is that the API wrapper classes may be used without MFC or OWL, but the event handling classes may not.

It is important to note while most of the WebLib APIs have been encapulated into C++ classes, some housekeeping and utility functions have not been, among them WLStartup() and WLCleanup(), which must be called on app startup and termination. The C API works fine for these functions.

MSVC++/MFC

Classes

The Microsoft Visual C++ API wrapper classes are CWeblibBrowserAPI, CWeblibToolbarAPI and CWeblibHtmlAPI. The event handling classes are CWeblibBrowser, CWeblibToolbar and CWeblibHtml.

The CWeblibBrowserAPI class parallels the C API in terms of functionality. One difference is CWeblibBrowserAPI provides an overloaded constructor that automatically connects to the Web browser when an instance of the class is created. After the object has been constructed, it is important to call the IsGood() method to determine if everything went OK. Upon destruction, the CWeblibBrowserAPI class will disconnect from the Web browser if needed.

The CWeblibBrowser class provides a virtual method for each event notification code that can be generated from any browser API. To handle an event, override the appropriate virtual method in your derived class.

The CWeblibToolbarAPI class provides a wrapper around the toolbar API and an overloaded constructor that will automatically create a toolbar. Call IsGood() to make sure a class instance was constructed correctly. The destructor deletes the toolbar if needed. The CWeblibToolbar class has a single virtual notification method that derived classes should override to handle button clicks.

The CWeblibHtmlAPI and CWeblibHtml classes are similar in design to the browser and toolbar classes. The CWeblibHtmlAPI has two overloaded constructors, one for parsing an HTML file and one for parsing an HTML buffer. The IsGood() method indicates whether the constructor successfully parsed the HTML or

not. If needed, WLEndParse() is called automatically when the class instance is destroyed. CWeblibHtml contains a virtual notification method for each enumeration API which derived classes should override to perform enumerations. Each enumeration method in CWeblibHtml takes an *enumeration ID*, which is passed to the virtual notification method. Since nested enumeration calls of the same type invoke the same virtual method, the enumeration ID provides a way to distinguish which enumeration is currently being handled.

Files

To use the CWeblibBrowserAPI, CWeblibBrowser, CWeblibToolbarAPI or CWeblibToolbar classes in an MSVC++ application, include the file **WEBLIBVC.H** in the source code and link the app with either **WEBLIVCS.LIB** or **WEBLIVCD.LIB**. The app must also be linked with **WEBLIB.LIB**.

To use the CWeblibHtmlAPI or CWeblibHtml classes, include the file **HTMLPAVC.H** in the source and link the app with either **HTMLPVCS.LIB** or **HTMLPVCD.LIB**. In addition, the app must be linked with **HTMLPARS.LIB**.

What's the difference between **WEBLIVCS.LIB** vs. **WEBLIVCD.LIB** and **HTMLPVCS.LIB** vs. **HTMLPVCD.LIB**? Which library you use depends upon whether your MSVC++ application statically links with MFC or uses the AFXDLL version of MFC. For statically linked apps, use **WEBLIVCS.LIB** and **HTMLPVCS.LIB**. Applications employing the AFXDLL version of MFC should link with **WEBLIVCD.LIB** and **HTMLPVCD.LIB**.

Compiling and Linking

Compile MSVC++ applications that use the WebLib class libraries with the large memory model and with a stack of at least 24K.

Borland C++/OWL

Classes

The Borland C++ API classes are TWeblibBrowserAPI, TWeblibToolbarAPI and TWeblibHtmlAPI and the event handling classes are TWeblibBrowser, TWeblibToolbar and TWeblibHtml. These classes are identical in functionality to the MSVC++/MFC classes.

Files

To use the TWeblibBrowserAPI, TWeblibBrowser, TWeblibToolbarAPI or TWeblibToolbar classes in a Borland C++ app, include the file **WEBLIBBC.H** in the source code and link the app with one of the **WEBLXXXX** libraries listed below. The app must also be linked with **WEBLIB.LIB**.

To use the TWeblibHtmlAPI or TWeblibHtml classes, include the file **HTMLPABC.H** in the source and link the app with one of the following **HTMLXXXX** libraries. In addition, the app must be linked with **HTMLPARS.LIB**.

Which library you link with depends on the type of module you are writing and which version of OWL you want to use:

WEBLBCDD.LIB	For writing DLLs that link with the DLL-based version of OWL
WEBLBCDE.LIB	For writing EXEs that link with the DLL-based version of OWL
WEBLBCSD.LIB	For writing DLLs that link with the static version of OWL
WEBLBCSE.LIB	For writing EXEs that link with the static version of OWL
HTMLBCDD.LIB	For writing DLLs that link with the DLL-based version of OWL
HTMLBCDE.LIB	For writing EXEs that link with the DLL-based version of OWL
HTMLBCSD.LIB	For writing DLLs that link with the static version of OWL

HTMLBCSE.LIB For writing EXEs that link with the static version of OWL

Compiling and Linking

As with MSVC++ apps, compile Borland C++ apps that use the WebLib class libraries with the large memory model and with a stack of 24K or more.

Delphi Overview

WebLib supports Delphi via a set of wrappers around the C API. While Delphi's VBX support is robust, conversion between Delphi's VBX data types and Object Pascal data types is a problem, especially when dealing with strings. This being the case, the WebLib VBXs are currently not supported under Delphi.

WebLib's C API is supported in Delphi by two program units: Weblib and Htmlpars. The source code for these units is stored in the files **WEBLIBDP.PAS** and **HTMLPADP.PAS**. Include these files in your project and reference the units as needed with the Object Pacal *uses* clause. For example:

```
program Project1;

uses
      Forms, WinTypes, WinProcs,
      Weblib in 'WEBLIBDP.PAS',
      Htmlpars in 'HTMLPADP.PAS',
      Unit1 in 'UNIT1.PAS' {Form1};
```

You may either copy **WEBLIBDP.PAS** and **HTMLPADP.PAS** into your Delphi project's directory or compile these files in the **WEBLIB\INCLUDE** directory and then add **WEBLIB\INCLUDE** to the Search Path in the Project Options dialog.

Initialization

WebLib must be initialized before using any of the browser or toolbar APIs. To do this, call WLStartup() from the application's entry point. To terminate WebLib, call WLCleanup(). For example:

```
program Project1;

uses
      ... ;

{$R *.RES}

begin
      if not WLStartup then
            Application.MessageBox('WLStartup','Error',MB_OK);

      Application.CreateForm(TForm1, Form1);
      Application.Run;

      if not WLCleanup then
            Application.MessageBox('WLCleanup','Error',MB_OK);
end.
```

Handling Events

Events are handled as they are in the C API, by sending a Windows message to a particular window. In Delphi, this involves using the **Handle** form attribute to obtain a window handle and creating a message-handling procedure with the **message** directive.

The pre-defined message identifier WM_WEBLIB_NOTIFY is included for use with the browser and toolbar APIs. In addition, the message identifiers WM_WEBLIB_ENUMPARSETREE, WM_WEBLIB_ENUMFINDTEXT, WM_WEBLIB_ENUMFINDSPECIAL, WM_WEBLIB_ENUMFINDCOMMENT, WM_WEBLIB_ENUMFINDTAGTYPE,

WM_WEBLIB_ENUMFINDTAGNAME, WM_WEBLIB_ENUMFINDTAGATTR have been pre-defined for use with the HTML parsing API. These message identifiers have been assigned in such a way that there should not be any conflict with any standard controls or windows.

Here is an example of how to setup and receive URLEcho events:

```
	{ ===== do registration for event ===== }

	if WLRegisterURLEcho(hBrowser,Handle,WM_WEBLIB_NOTIFY) = 0 then
		Application.MessageBox('RegisterEchoURL failed','Error',MB_OK);

	{ ===== declaration of message handling prodecure ===== }

	type
	TForm1 = class(TForm)
		{ declaration of controls, etc. }
	private
		{ Private declarations }
	public
		procedure HandleEvent(var msg: TMessage); message
WM_WEBLIB_NOTIFY;
	end;

	{ ===== implementation variables ===== }

	var
		hBrowser: Weblib.HBROWSER;
		szURL: array[0..255] of Char;
		szMIME: array[0..255] of Char;
		szRef: array[0..255] of Char;
		dwWindow: LongInt;

	{ ===== implementation of message handing procedure ===== }

	procedure TForm1.HandleEvent (var msg: TMessage);
	begin
		if msg.wParam = WLN_URLECHO then
		begin
			dwWindow := WLNGetWindow(msg.LParam);
			szURL := WLNGetURL(msg.LParam);
			szMIME := WLNGetMIMEType(msg.LParam);
			szRef := WLNGetReferrer(msg.LParam);

			{ ...do something useful here... }
		end;
	end;
```

The enumeration functions in the HTML parsing API require that TRUE be returned from the message handler to continue the enumeration. To do this, set the **Result** member of **TMessage**.

Strings

Zero-terminated strings must always be used with the WebLib APIs. In Object Pascal, such strings are declared using the following syntax:

```
	var
```

```
    szURL: array[0..255] of Char;
```

The **PChar** data type and various **Str**... functions (e.g., **StrCopy**) are also useful when working with zero-terminated strings.

Adding Toolbar Buttons

The WebLib extension APIs include several functions for adding a button to a toolbar that are easier to use with Delphi than WLAddToobarButton(). In particular, the extension function **WLAddToobarButtonByFile**() works well with Delphi since it loads bitmaps from files. The bitmaps stored in the files must be DIB-compatible but not compressed (this is usually the case with files having the .BMP extension).

Startup API

Boolean Startup()

Initializes the WebLib DLL and must be called before any WebLib browser or toolbar API.

Return Value

TRUE if successful, FALSE if an error occurred.

See Also

Cleanup

C/C++

```
BOOL    WLStartup();
```

Visual Basic

```
Function weblibStartup() As Integer
```

Cleanup API

Boolean Cleanup()

Shuts down the WebLib DLL. Do not call WebLib browser or toolbar APIs after **Cleanup**() has been called.

Return Value

TRUE if successful, FALSE if an error occurred.

See Also

Startup

C/C++

```
BOOL    WLCleanup();
```

Visual Basic

```
Function weblibCleanup() As Integer
```

SetDefaultNotify API

Boolean SetDefaultNotify(*NotifyWindow***,***Message***)**

Sets the default notification window and message. Other WebLib API calls that take a window and message as parameters may reference *NotifyWindow* and *Message* by passing WL_DEFAULTNOTIFY in place of a window (message is ignored is this case and may be set to zero).

NotifyWindow will also receive BrowserStart and BrowserExit events.

Arguments

NotifyWindow	Handle of window to receive events.
Message	Notification message.

Return Value

TRUE if successful, FALSE if an error occurred.

C/C++

BOOL WLSetDefaultNotify(HWND *hwndNotify*,UINT *nMsg*);

Visual Basic

Not currently supported

ConnectBrowser API

Browser ConnectBrowser(*Options***)**

Establishes logical connection with Web browser.

Arguments

Options	WL_STARTBROWSER starts the Web browser if it is not already running. Other options are WL_POSTMESSAGE (notify of events via **PostMessage**) and WL_SENDMESSAGE (notify via **SendMessage**).

Return Value

Handle to browser object.

See Also

DisconnectBrowser

C

HBROWSER WLConnectBrowser(WORD *wOptions*);

C++

BOOL CWeblibBrowser::ConnectBrowser(WORD *wOptions*=WL_POSTMESSAGE|
WL_STARTBROWSER);

VBX

WLBrowser.Action = actionConnectBrowser(ByVal *wOptions* As Integer)

DisconnectBrowser API

Boolean DisconnectBrowser(*ExitBrowser***)**

Closes logical connection with Web browser.

Arguments

ExitBrowser	If TRUE and there are no other open connections to the Web browser, the browser is requested to exit.

Return Value

TRUE if successful, FALSE if an error occurred.

See Also

ConnectBrowser

C

BOOL WLDisconnectBrowser(HBROWSER *hBrowser*,BOOL *bExit*);

C++

BOOL CWeblibBrowser::DisconnectBrowser(BOOL *bExit*=TRUE);

VBX

WLBrowser.Action = actionDisconnectBrowser(ByVal *bExit* As Integer)

GetWindowInfo API

Boolean GetWindowInfo(*WindowID***,***URL***,***Title***)**

Returns the URL and title associated with a browser window.

Arguments

WindowID	Browser window to get URL and Title for or WL_ACTIVEWINDOW to get the URL and Title of the active browser window.
URL	URL of browser window is copied here.
Title	Title of browser window is copied here.

Return Value

TRUE if successful, FALSE if an error occurred.

Netscape

This function fails if either URL or Title is not available.

C

BOOL WLGetWindowInfo(HBROWSER *hBrowser*,DWORD *dwWindow*,LPSTR *lpszURL*,int *cbURL*,LPSTR *lpszTitle*,int *cbTitle*);

C++

BOOL CWeblibBrowser::GetWindowInfo(DWORD *dwWindow*,LPSTR *lpszURL*,int *cbURL*,LPSTR *lpszTitle*,int *cbTitle*) const;

VBX

WLBrowser.Action = actionGetWindowInfo(ByVal *dwWindow* As Long)

Notes

Sets the properties URL and Title if successful.

ListWindows API

WindowID ListWindows(*GetFirst***)**

Enumerates Web browser windows.

Arguments

GetFirst	If TRUE, returns the first browser window, otherwise returns the next browser window.

Return Value

Next browser window if successful or zero if there are no more browser windows or if an error occurred.

C

DWORD WLListWindows(HBROWSER *hBrowser*,BOOL *bFirst*);

C++

DWORD CWeblibBrowser::ListWindows(BOOL *bFirst*) const;

VBX

WLBrowser.Action = actionListWindows()

Notes

Enumerates all browser windows in one call, setting the WindowList property array. This array is delimited by a zero window ID.

ActivateWindow API

WindowID ActivateWindow(*WindowID***)**

Activates a browser window.

Arguments

WindowID	Browser window to activate or WL_LASTACTIVEWINDOW to re-activate the last active window.

Return Value

The browser window activated if successful or zero if an error occurred.

C

DWORD WLActivateWindow(HBROWSER *hBrowser*,DWORD *dwWindow*);

C++

DWORD CWeblibBrowser::ActivateWindow(DWORD *dwWindow*) const;

VBX

WLBrowser.Action = actionActivateWindow(ByVal *dwWindow* As Long)

Notes

Sets the Window property if successful.

CloseWindow API

Boolean CloseWindow(*WindowID***)**

Closes a browser window.

Arguments

WindowID	Browser window to close.

Return Value

TRUE if successful or FALSE if an error occurred.

C

BOOL WLCloseWindow(HBROWSER *hBrowser*,DWORD *dwWindow*);

C++

BOOL CWeblibBrowser::CloseWindow(DWORD *dwWindow*) const;

VBX

WLBrowser.Action = actionCloseWindow(ByVal *dwWindow* As Long)

SetWindowPos API

Boolean SetWindowPos(*WindowID***,***X***,***Y***,***Width***,***Height***)**

Sets the position and size of a browser window.

Arguments

WindowID	Browser window to size or move.
X	New X coordinate of upper left corner of window.
Y	New Y coordinate of upper left corner of window.
Width	New window width or WL_NOCHANGE to retain current width.
Height	New window height or WL_NOCHANGE to retain current height.

Return Value

TRUE if successful or FALSE if an error occurred.

Netscape

Requires *X*, *Y*, *Width* and *Height* (does not respect WL_NOCHANGE).

C

BOOL WLSetWindowPos(HBROWSER *hBrowser*,DWORD *dwWindow*,DWORD *dwX*,DWORD *dwY*,DWORD *dwWidth*,DWORD *dwHeight*);

C++

BOOL CWeblibBrowser::SetWindowPos(DWORD *dwWindow*,DWORD *dwX*,DWORD *dwY*,DWORD *dwWidth*,DWORD *dwHeight*) const;

VBX

WLBrowser.Action = actionSetWindowPos(ByVal *dwWindow* As Long, ByVal *dwX* As Long, ByVal *dwY* As Long, ByVal *dwWidth* As Long, ByVal *dwHeight* As Long)

ShowWindow API

Boolean ShowWindow(*WindowID***,***ShowFlag***)**

Minimizes, maximizes or restores a browser window.

Arguments

WindowID	Browser window to minimize, maximize or restore.
ShowFlag	WL_MINIMIZE, WL_MAXIMIZE or WL_NORMAL.

Return Value

TRUE if successful or FALSE if an error occurred.

C

BOOL WLShowWindow(HBROWSER *hBrowser*,DWORD *dwWindow*,UINT *nShow*);

C++

BOOL CWeblibBrowser::ShowWindow(DWORD *dwWindow*,UINT *nShow*) const;

VBX

WLBrowser.Action = actionShowWindow(ByVal *dwWindow* As Long, ByVal *nShow* As Integer)

ShowFile API

Transaction ShowFile(*File***,***MIME***,***WindowID***,***URL***,***NotifyWindow***,***Message***)**

Displays a file in a browser window, sending progress events to *NotifyWindow*.

Arguments

File	Name of local file to display.
MIME	MIME type of file contents.
WindowID	Browser window to display file in or WL_NEWWINDOW to open a new browser window.
URL	URL of file in case it needs to re-loaded.
NotifyWindow	Handle of window to receive events or WL_NONOTIFY for silence.
Message	Notification message.

Return Value

Transaction ID if successful or zero if an error occurred.

C

DWORD WLShowFile(HBROWSER *hBrowser*,LPCSTR *lpszFile*,LPCSTR *lpszMIMEType*,DWORD *dwWindow*,LPCSTR *lpszURL*,HWND *hwndNotify*,UINT *nMsg*);

C++

DWORD CWeblibBrowser::ShowFile(LPCSTR *lpszFile*,LPCSTR *lpszMIMEType*,DWORD *dwWindow*,LPCSTR *lpszURL*,BOOL *bNotify*=TRUE) const;

VBX

WLBrowser.Action = actionShowFile(ByVal *lpszFile* As String, ByVal *lpszMIME* As String, ByVal *dwWindow* As Long, ByVal *lpszURL* As String)

Notes

Sets the Transaction property if successful.

OpenURL API

Transaction OpenURL(*URL*,*WindowID*,*Options*,*NotifyWindow*,*Message***)**

Opens a URL, sending progress events to *NotifyWindow*.

Arguments

URL	URL to open.
WindowID	Browser window to display URL in or WL_NEWWINDOW to open a new browser window.
Options	WL_NODOCUMENTCACHE (ignore document cache), WL_NOIMAGECACHE (ignore image cache) or WL_BACKGROUNDMODE (operate in background mode). These options may be combined.
NotifyWindow	Handle of window to receive events or WL_NONOTIFY for silence.
Message	Notification message.

Return Value

Transaction ID if successful or zero if an error occurred.

See Also

SaveURL

Netscape

The WL_NODOCUMENTCACHE, WL_NOIMAGECACHE and WL_BACKGROUNDMODE options are ignored.

C

DWORD WLOpenURL(HBROWSER *hBrowser*,LPCSTR *lpszURL*,DWORD *dwWindow*,WORD *wOptions*,HWND *hwndNotify*,UINT *nMsg*);

C++

DWORD CWeblibBrowser::OpenURL(LPCSTR *lpszURL*,DWORD *dwWindow*,WORD *wOptions*,BOOL *bNotify*=TRUE) const;

VBX

WLBrowser.Action = actionOpenURL(ByVal *lpszURL* As String, ByVal *dwWindow* As Long, ByVal *wOptions* As Integer)

Notes

Sets the Transaction property if successful.

SaveURL API

Transaction SaveURL(*URL***,***File***,***WindowID***,***Options***,***NotifyWindow***,***Message***)**

Opens a URL and saves the loaded document to a file. Also sends progress events to*NotifyWindow*.

Arguments

URL	URL to open.
File	Local file to save document in.
indowID	Browser window to display URL in or WL_NEWWINDOW to open a new browser window.
Options	WL_NODOCUMENTCACHE (ignore document cache), WL_NOIMAGECACHE (ignore image cache) or WL_BACKGROUNDMODE (operate in background mode). These options may be combined.
NotifyWindow	Handle of window to receive events or WL_NONOTIFY for silence.
Message	Notification message.

Return Value

Transaction ID if successful or zero if an error occurred.

See Also

OpenURL

Netscape

The WL_NODOCUMENTCACHE, WL_NOIMAGECACHE and WL_BACKGROUNDMODE options are ignored.

C

DWORD WLSaveURL(HBROWSER *hBrowser*,LPCSTR *lpszURL*,LPCST *lpszFile*,DWORD *dwWindow*,WORD *wOptions*,HWND *hwndNotify*,UINT nMsg);

C++

DWORD CWeblibBrowser::SaveURL(LPCSTR *lpszURL*,LPCSTR *lpszFile*,DWORD *dwWindow*,WORD *wOptions*,BOOL *bNotify*=TRUE) const;

VBX

WLBrowser.Action = actionSaveURL(ByVal *lpszURL* As String,ByVal *lpszFile* As String, ByVal *dwWindow* As Long, ByVal *wOptions* As Integer)

Notes

Sets the Transaction property if successful.

PostFormData API

Transaction PostFormData(URL,WindowID,FormData,MIME,NotifyWindow,Message)

Sends *FormData* to *URL* using the HTTP POST method and displays the result in a browser window, sending progress events to *NotifyWindow*.

Arguments

URL	URL to post data to.
WindowID	Browser window to display result in or WL_NEWWINDOW to open a new browser window.
FormData	Data buffer to send, usually a url-encoded set of variables and values similar to the output produced by Web forms.
MIME	MIME type of form data.
NotifyWindow	Handle of window to receive events or WL_NONOTIFY for silence.
Message	Notification message.

Return Value

Transaction ID if successful or zero if an error occurred.

See Also

SaveFormData,AppendFormData,AccessFormData

C

DWORD WLPostFormData(HBROWSER *hBrowser*,LPCSTR *lpszURL*,DWORD *dwWindow*,LPCSTR *lpszFormData*,LPCSTR *lpszMIMEType*,HWND *hwndNotify*,UINT *nMsg*);

C++

DWORD CWeblibBrowser::PostFormData(LPCSTR *lpszURL*,DWORD *dwWindow*,LPCSTR *lpszFormData*,LPCSTR *lpszMIMEType*,BOOL *bNotify*=TRUE) const;

VBX

WLBrowser.Action = actionPostFormData(ByVal *lpszURL* As String, ByVal *dwWindow* As Long, ByVal *lpszFormData* As String, ByVal *lpszMIMEType* As String)

Notes

Sets the Transaction property if successful.

SaveFormData API

Transaction SaveFormData(*URL***,***File***,***WindowID***,***FormData***,***MIME***,***NotifyWindow***,***Message***)**

Sends *FormData* to *URL* using the HTTP POST method and saves the result in file. Also sends progress events to *NotifyWindow*.

Arguments

URL	URL to post data to.
File	Local file to save result in.
WindowID	Browser window to display result in or WL_NEWWINDOW to open a new browser window.
FormData	Data buffer to send, usually a url-encoded set of variables and values similar to the output produced by Web forms.
MIME	MIME type of form data.
NotifyWindow	Handle of window to receive events or WL_NONOTIFY for silence.
Message	Notification message.

Return Value

Transaction ID if successful or zero if an error occurred.

See Also

PostFormData,AppendFormData,AccessFormData

C

DWORD WLSaveFormData(HBROWSER *hBrowser*,LPCSTR *lpszURL*,LPCSTR *lpszFile*,DWORD *dwWindow*,LPCSTR *lpszFormData*,LPCSTR *lpszMIMEType*,HWND *hwndNotify*,UINT *nMsg*);

C++

DWORD CWeblibBrowser::SaveFormData(LPCSTR *lpszURL*,LPCSTR *lpszFile*,DWORD *dwWindow*,LPCSTR *lpszFormData*,LPCSTR *lpszMIMEType*,BOOL *bNotify*=TRUE) const;

VBX

WLBrowser.Action = actionSaveFormData(ByVal *lpszURL* As String, ByVal *lpszFile* As String,ByVal *dwWindow* As Long, ByVal *lpszFormData* As String, ByVal *lpszMIMEType* As String)

Notes

Sets the Transaction property if successful.

Cancel API

Boolean Cancel(*Transaction***)**

Cancels a Transaction, sending the Canceled event.

Arguments

Transaction	Transaction to cancel.

Return Value

TRUE if successful or FALSE if an error occurred.

See Also

ShowFile,OpenURL,SaveURL,PostFormData,SaveFormData

C

BOOL WLCancel(HBROWSER *hBrowser*,DWORD *dwTransaction*);

C++

BOOL CWeblibBrowser::Cancel(DWORD *dwTransaction*) const;

VBX

WLBrowser.Action = actionCancel(ByVal *dwTransaction* As Long)

RegisterProtocol API

Boolean RegisterProtocol(*Protocol*,*NotifyWindow*,*Message***)**

Registers app as a handler for *Protocol*. A OpenURL event is sent to *NotifyWindow* when the Web browser encounters a URL that uses *Protocol*.

Arguments

Protocol	Protocol that app wants to handle. This may be a standard protocol such as 'HTTP' or a custom, user-defined protocol.
NotifyWindow	Handle of window to receive events.
Message	Notification message.

Return Value

TRUE if successful or FALSE if an error occurred.

See Also

UnregisterProtocol

C

BOOL WLRegisterProtocol(HBROWSER *hBrowser*,LPCSTR *lpszProtocol*,HWND *hwndNotify*,UINT *nMsg*);

C++

BOOL CWeblibBrowser::RegisterProtocol(LPCSTR *lpszProtocol*) const;

VBX

WLBrowser.Action = actionRegisterProtocol(ByVal *lpszProtocol* As String)

UnregisterProtocol API

Boolean UnregisterProtocol(*Protocol***,***NotifyWindow***)**

Unregisters app as a handler for *Protocol*. OpenURL events will no longer be sent to *NotifyWindow* when the Web browser encounters a URL that uses *Protocol*.

Arguments

Protocol	Protocol that app no longer wants to handle.
NotifyWindow	Handle of window that was receiving events.

Return Value

TRUE if successful or FALSE if an error occurred.

See Also

RegisterProtocol

C

BOOL WLUnregisterProtocol(HBROWSER *hBrowser*,LPCSTR *lpszProtocol*,HWND *hwndNotify*);

C++

BOOL CWeblibBrowser::UnregisterProtocol(LPCSTR *lpszProtocol*) const;

VBX

WLBrowser.Action = actionUnregisterProtocol(ByVal *lpszProtocol* As String)

RegisterURLEcho API

Boolean RegisterURLEcho(*NotifyWindow***,***Message***)**

Registers app to receive URLEcho events whenever the Web browser opens a URL. The event is sent to *NotifyWindow*.

Arguments

NotifyWindow	Handle of window to receive events.
Message	Notification message.

Return Value

TRUE if successful or FALSE if an error occurred.

See Also

UnregisterURLEcho

C

BOOL WLRegisterURLEcho(HBROWSER *hBrowser*,HWND *hwndNotify*,UINT *nMsg*);

C++

BOOL CWeblibBrowser::RegisterURLEcho() const;

VBX

WLBrowser.Action = actionRegisterURLEcho()

UnregisterURLEcho API

Boolean UnregisterURLEcho(*NotifyWindow***)**

Unregisters app from receiving URLEcho events. Events are no longer sent to *NotifyWindow*.

Arguments

NotifyWindow	Handle of window that was receiving events.

Return Value

TRUE if successful or FALSE if an error occurred.

See Also

RegisterURLEcho

C

BOOL WLUnregisterURLEcho(HBROWSER *hBrowser*,HWND *hwndNotify*);

C++

BOOL CWeblibBrowser::UnregisterURLEcho() const;

VBX

WLBrowser.Action = actionUnegisterURLEcho()

RegisterViewer API

Boolean RegisterViewer(*MIME***,***Options***,***NotifyWindow***,***Message***)**

Registers app as a viewer for documents of a certain type. Depending upon *Options*,a QueryViewer and/or a ViewDocFile event are sent to *NotifyWindow* when the Web browser loads a document of type *MIME*.

Arguments

MIME	MIME type to act as viewer for.
Options	WL_SHELLEXECUTE (the browser calls ShellExecute on the document file), WL_QUERYVIEWER (the browser sends the QueryViewer event to obtain a filename to save the document in) or WL_VIEWDOCFILE (the browser sends the ViewDocFile event to ask viewer to display a document). These values may be combined.
NotifyWindow	Handle of window to receive events.
Message	Notification message.

Return Value

TRUE if successful or FALSE if an error occurred.

See Also

UnregisterViewer,SetFileName

C

BOOL WLRegisterViewer(HBROWSER *hBrowser*,LPCSTR *lpszMIMEType*,WORD *wOptions*,HWND *hwndNotify*,UINT *nMsg*);

C++

BOOL CWeblibBrowser::RegisterViewer(LPCSTR *lpszMIMEType*,WORD *wOptions*) const;

VBX

WLBrowser.Action = actionRegisterViewer(ByVal *lpszMIMEType* As String, ByVal *wOptions* As Integer)

UnregisterViewer API

Boolean UnregisterViewer(*MIME***,***NotifyWindow***)**

Unregisters app as a viewer for documents of type *MIME*. QueryViewer and ViewDocFile events are no longer sent to *NotifyWindow*.

Arguments

MIME	MIME type to stop acting as viewer for.
NotifyWindow	Handle of window that was receiving events.

Return Value

TRUE if successful or FALSE if an error occurred.

See Also

RegisterViewer

C

BOOL WLUnregisterViewer(HBROWSER *hBrowser*,LPCSTR *lpszMIMEType*,HWND *hwndNotify*);

C++

BOOL CWeblibBrowser::UnregisterViewer(LPCSTR *lpszMIMEType*) const;

VBX

WLBrowser.Action = actionUnregisterViewer(ByVal *lpszMIMEType* As String)

RegisterWindowChange API

Boolean RegisterWindowChange(*WindowID***,***NotifyWindow***,***Message***)**

Registers app to receive WindowChange events for a browser window.

Arguments

WindowID	Browser window to monitor for changes.
NotifyWindow	Handle of window to receive events.
Message	Notification message.

Return Value

TRUE if successful or FALSE if an error occurred.

See Also

UnregisterWindowChange,SetWindowPos

C

BOOL WLRegisterWindowChange(HBROWSER *hBrowser*,DWORD *dwWindow*,HWND *hwndNotify*,UINT *nMsg*);

C++

BOOL CWeblibBrowser::RegisterWindowChange(DWORD *dwWindow*) const;

VBX

WLBrowser.Action = actionRegisterWindowChange(ByVal *dwWindow* As Long)

UnregisterWindowChange API

Boolean UnregisterWindowChange(*WindowID***,***NotifyWindow***)**

Unregisters app from receiving WindowChange events for a browser window.

Arguments

WindowID	Browser window to stop monitoring for changes.
NotifyWindow	Handle of window that was receiving events.

Return Value

TRUE if successful or FALSE if an error occurred.

See Also

RegisterWindowChange

C

BOOL WLUnregisterWindowChange(HBROWSER *hBrowser*,DWORD *dwWindow*,HWND *hwndNotify*);

C++

BOOL CWeblibBrowser::UnregisterWindowChange(DWORD *dwWindow*) const;

VBX

WLBrowser.Action = actionUnregisterWindowChange(ByVal dwWindow As Long)

ParseAnchor API

URL ParseAnchor(*AbsoluteURL***,***RelativeURL***)**

Combines an absolute URL and a relative URL to form a new absolute URL.

Arguments

AbsoluteURL	Base URL, must be absolute.
RelativeURL	URL to combine with base URL, must be relative.

Return Value

The combination of *AbsoluteURL* and *RelativeURL* or NULL if an error occurred.

C

LPCSTR WLParseAnchor(HBROWSER *hBrowser*,LPCSTR *lpszAbsoluteURL*,LPCSTR *lpszRelativeURL*);

C++

LPCSTR CWeblibBrowser::ParseAnchor(LPCSTR *lpszAbsoluteURL*,LPCSTR *lpszRelativeURL*) const;

VBX

WLBrowser.Action = actionParseAnchor(ByVal *lpszAbsoluteURL* As String, ByVal *lpszRelativeURL* As String)

Notes

Sets the URL property if successful.

GetVersion API

VersionInfo GetVersion(*Major***,***Minor***)**

Returns the highest version of the Web browser's DDE interface that is compatible with *Major* and *Minor* (i.e., less than or equal to *Major* and *Minor*).

Arguments

AbsoluteURL	Base URL, must be absolute.
RelativeURL	URL to combine with base URL, must be relative.

Return Value

Major (high word) and Minor (low word) version of DDE interface supported by the Web browser or zero if an error occurred.

Netscape

Major and *Minor* are ignored and can be any value.

C

DWORD WLGetVersion(HBROWSER *hBrowser*,WORD *wMajor*,WORD *wMinor*);

C++

DWORD CWeblibBrowser::GetVersion(WORD *wMajor*,WORD *wMinor*) const;

VBX

WLBrowser.Action = actionGetVersion(ByVal *wMajor* As Integer, ByVal *wMinor* As Integer)

Notes

Sets the MajorVersion and MinorVersion properties if successful.

QueryURLFile API

URL QueryURLFile(*File*)

Returns the URL that *File* was loaded from. This is a specialized API for applications that handle document viewing.

Arguments

File	Local file currently displayed in a browser window.

Return Value

URL associated with *File* or NULL if an error occurred.

Enhanced Mosaic

This API is a no-op since it is Netscape-specific.

C

LPCSTR WLQueryURLFile(HBROWSER *hBrowser*,LPCSTR *lpszFile*);

C++

LPCSTR CWeblibBrowser::QueryURLFile(LPCSTR *lpszFile*) const;

VBX

WLBrowser.Action = actionQueryURLFile(ByVal *lpszFile* As String)

Notes

Sets the URL property if successful.

SetNotifyMethod API

Boolean SetNotifyMethod(*Method***)**

Sets the method used to notify an application of events.

Arguments

Method	WL_POSTMESSAGE (recommended) or WL_SENDMESSAGE.

Return Value

TRUE if successful or FALSE if an error occurred.

C

BOOL WLSetNotifyMethod(HBROWSER *hBrowser*,WORD *wMethod*);

C++

BOOL CWeblibBrowser::SetNotifyMethod(WORD *wMethod*) const;

VBX

WLBrowser.Action = actionSetNotifyMethod(ByVal *wMethod* As Integer)

GetTransactionWindow API

WindowID GetTransactionWindow(*Transaction***)**

Returns the browser window associated with a transaction ID. Once the browser window ID is returned, the entry for *Transaction* is removed from an internal cache; subsequent calls to **GetTransactionWindow**() using same transaction ID will fail.

The Finished event may also be used to obtain the browser window associated the with a transaction.

Arguments

Transaction	Transaction ID to return window for.

Return Value

Window ID if successful, 0xFFFFFFFF if window ID is not available yet or zero there is no window ID associated with *Transaction*.

See Also

OpenURL,SaveURL,PostFormData,SaveFormData,ShowFile,Finished event

Netscape

The browser window is available immediately after calling OpenURL or another API that returns a transaction ID.

Enhanced Mosaic

The browser window is not avaiable until the URL is completely loaded. Either grab the window ID from the Finished Event or call **GetTransactionWindow**() in a loop until it returns zero or a valid transaction ID (i.e., try again if 0xFFFFFFFF is returned).

C

DWORD WLGetTransactionWindow(HBROWSER *hBrowser*,DWORD *dwTransaction*);

C++

DWORD CWeblibBrowser::GetTransactionWindow(DWORD *dwTransaction*) const;

VBX

WLBrowser.Action = actionGetTransactionWindow(ByVal *dwTransaction* As Long)

SetFileName API

SetFileName(*FileName***)**

Sets the filename used to store a document that the application is responsible for displaying. Only call this function in response to the QueryViewer event.

Arguments

FileName	File to save document in. This should be the name of a file that does not already exist.

Return Value

None.

See Also

QueryViewer event.

C

void WLNSetFileName(LPARAM *lParam*,LPCSTR *lpszFileName*);

C++

Not needed since filename can be specified when handling event

VBX

Not needed since filename can be specified when handling event

CreateToolbar API

Toolbar CreateToolbar(Menu,NotifyWindow,Message)

Creates a toolbar that attaches itself to the Web browser.

Arguments

Menu	Text describing toolbar that appears in the toolbar's popup menu.
NotifyWindow	Handle of window to receive events.
Message	Notification message.

Return Value

Handle to toolbar object.

See Also

DeleteToolbar

C

HTOOLBAR WLCreateToolbar(LPCSTR *lpszMenu*,HWND *hwndNotify*,UINT *nMsg*);

C++

BOOL CWeblibToolbar::CreateToolbar(LPCSTR *lpszMenu*);

VBX

WLToolbar.Action = actionCreateToolbar(ByVal *szMenuText* As String)

Notes

Upon creation, the toolbar is initialized based on any design-time properties that have been set. If the AutoCreate property is True, there is no need to call **actionCreateToolbar**().

DeleteToolbar API

Boolean DeleteToolbar()

Deletes a toolbar.

Return Value

TRUE if successful or FALSE if an error occurred.

See Also

CreateToolbar

C

BOOL WLDeleteToolbar(HTOOLBAR *hToolbar*);

C++

BOOL CWeblibToolbar::DeleteToolbar();

VBX

WLToolbar.Action = actionDeleteToolbar()

SetActiveToolbar API

Boolean SetActiveToolbar()

Make this toolbar the active toolbar.

Return Value

TRUE if successful or FALSE if an error occurred.

See Also

IsToolbarActive

C

```
BOOL WLSetActiveToolbar(HTOOLBAR hToolbar);
```

C++

```
BOOL CWeblibToolbar::SetActiveToolbar() const;
```

VBX

```
WLToolbar.Action = actionSetActiveToolbar()
```

IsToolbarActive API

Boolean IsToolbarActive()

Determines if this toolbar is the active toolbar.

Return Value

TRUE if toolbar is active or FALSE if it is not.

See Also

SetActiveToolbar

C

BOOL WLIsToolbarActive(HTOOLBAR *hToolbar*);

C++

BOOL CWeblibToolbar::IsToolbarActive() const;

VBX

WLToolbar.Action = actionIsToolbarActive()

AddToolbarButton API

Boolean AddToolbarButton(*ButtonID***,** *ButtonIdx***,** *Text***,** *BitmapUp***,** *BitmapSelected***,** *BitmapFocus***,** *BitmapDisabled***)**

Adds a bitmap button to a toolbar. Each bitmap must be 24 x 24 pixels in size.

Arguments

ButtonID	Unique identifier for button.
ButtonIdx	Position of button starting from zero (leftmost position).
Text	Text to display when cursor is placed over button.
BitmapUp	Bitmap to display when button is up (required).
BitmapSelected	Bitmap to display when button is pressed (optional).
BitmapFocus	Bitmap to display when button has focus (optional).
BitmapDisabled	Bitmap to display when button is disabled (optional).

Return Value

TRUE if successful or FALSE if an error occurred.

See Also

RemoveToolbarButton,**Weblib extension APIs**

C

BOOL WINAPI WLAddToolbarButton(HTOOLBAR *hToolbar*,UINT *nID*,int *nIdx*,LPCSTR *lpszText*, HINSTANCE *hInstance*,LPCSTR *lpszBitmap*,LPCSTR *lpszBitmapSel*,LPCSTR *lpszBitmapFocus*, LPCSTR *lpszBitmapDisabled*);

Notes

This function can create a button by bitmap handle, integer resource ID or string resource ID. To create a button using bitmap handles, pass WL_BITMAPHANDLES through *hInstance* instead of an instance handle and pass HBITMAPs through *lpszBitmap*, *lpszBitmapSel*,*lpszBitmapFocus* and *lpszBitmapDisabled*. To pass integer resource IDs, use the **MAKEINTRESOURCE** macro.

C++

BOOL CWeblibToolbar::AddToolbarButton(UINT *nID*,int *nIdx*,LPCSTR *lpszText*,HINSTANCE *hInstance*,LPCSTR *lpszBitmap*,LPCSTR *lpszBitmapSel*,LPCSTR *lpszBitmapFocus*,LPCSTR *lpszBitmapDisabled*) const;

Notes

This function can create a button by bitmap handle, integer resource ID or string resource ID. See the notes for **WLAddToolbarButton**().

VBX

WLToolbar.Action = actionAddToolbarButtonByHandle(ByVal *nID* As Integer, ByVal *nIdx* As Integer, ByVal *szText* As String,ByVal *bmUp* As Integer,ByVal *bmSel* As Integer,ByVal *bmFoc* As Integer,ByVal *bmDis* As Integer,ByVal *bCopy* As Integer)

WLToolbar.Action = actionAddToolbarButtonByID(ByVal *nID* As Integer, ByVal *nIdx* As Integer,ByVal *szText* As String,ByVal *nInst* As Integer,ByVal *bmUp* As Integer,ByVal *bmSel* As Integer,ByVal *bmFoc* As Integer,ByVal *bmDis* As Integer)

WLToolbar.Action = actionAddToolbarButtonByName(ByVal *nID* As Integer, ByVal *nIdx* As Integer, ByVal *szText* As String,ByVal *nInst* As Integer,ByVal *bmUp* As String,ByVal *bmSel* As String,ByVal *bmFoc* As String,ByVal *bmDis* As String)

Notes

The first function above creates a button based on bitmap handles (HBITMAP) and may be used with the picture box control's **Image** property. When used with the **Image** property, *bCopy* must be TRUE (copies the bitmap instead of using the original).

The latter two functions load bitmap resources from the executable or dynamic link library whose instance handle is *nInst*.

RemoveToolbarButton API

Boolean RemoveToolbarButton(*ButtonID***)**

Deletes a button from a toolbar.

Arguments

ButtonID	Button to delete.

Return Value

TRUE if successful or FALSE if an error occurred.

See Also

SetActiveToolbar

C

BOOL WLRemoveToolbarButton(HTOOLBAR *hToolbar*,UINT *nID*);

C++

BOOL CWeblibToolbar::RemoveToolbarButton(UINT *nID*) const;

VBX

WLToolbar.Action = actionRemoveToolbarButton(ByVal *nID* As Integer)

IsToolbarButtonVisible API

Boolean IsToolbarButtonVisible(*ButtonID***)**

Determines if toolbar button is visible.

Arguments

ButtonID	Button to check for visibility.

Return Value

TRUE if button is visible or FALSE if button is hidden.

See Also

ShowToolbarButton

C

BOOL WLIsToolbarButtonVisible(HTOOLBAR *hToolbar*,UINT *nID*);

C++

BOOL CWeblibToolbar::IsToolbarButtonVisible(UINT *nID*) const;

VBX

WLToolbar.Action = actionIsToolbarButtonVisible(ByVal *nID* As Integer)

ShowToolbarButton API

Boolean ShowToolbarButton(*ButtonID*,*ShowFlag***)**

Show or hides a toolbar button.

Arguments

ButtonID	Button to show or hide.
ShowFlag	TRUE to show button or FALSE to hide button.

Return Value

TRUE if successful or FALSE if an error occurred.

See Also

IsToolbarButtonVisible

C

BOOL WLShowToolbarButton(HTOOLBAR *hToolbar*,UINT *nID*,BOOL *bShow*);

C++

BOOL CWeblibToolbar::ShowToolbarButton(UINT *nID*,BOOL *bShow*) const;

VBX

WLToolbar.Action = actionShowToolbarButton(ByVal *nID* As Integer, ByVal *bShow* As Integer)

IsToolbarButtonEnabled API

Boolean IsToolbarButtonEnabled(*ButtonID***)**

Determines if toolbar button is enabled.

Arguments

ButtonID	Button to check enabled state for.

Return Value

TRUE if button is enabled or FALSE if button is disabled.

See Also

EnableToolbarButton

C

BOOL WLIsToolbarButtonEnabled(HTOOLBAR *hToolbar*,UINT *nID*);

C++

BOOL CWeblibToolbar::IsToolbarButtonEnabled(UINT *nID*) const;

VBX

WLToolbar.Action = actionIsToolbarButtonEnabled(ByVal *nID* As Integer)

EnableToolbarButton API

Boolean EnableToolbarButton(*ButtonID***,***EnableFlag***)**

Enables or disables a toolbar button.

Arguments

ButtonID	Button to show or hide.
EnableFlag	TRUE to enable button or FALSE to disable button.

Return Value

TRUE if successful or FALSE if an error occurred.

See Also

IsToolbarButtonEnabled

C

BOOL WLEnableToolbarButton(HTOOLBAR *hToolbar*,UINT *nID*,BOOL *bEnable*);

C++

BOOL CWeblibToolbar::EnableToolbarButton(UINT *nID*,BOOL *bEnable*) const;

VBX

WLToolbar.Action = actionEnableToolbarButton(ByVal *nID* As Integer, ByVal *bEnable* As Integer)

GetToolbarText API

String GetToolbarText(*ButtonID***)**

Returns the text for a toolbar or toolbar button.

Arguments

ButtonID	Button to get text for or WL_TOOLBARTEXT to get toolbar's text.

Return Value

Text for button or toolbar if successful or NULL if an error occurred.

See Also

SetToolbarText

C

LPCSTR WLGetToolbarText(HTOOLBAR *hToolbar*,UINT *nID*);

C++

LPCSTR CWeblibToolbar::GetToolbarText(UINT *nID*) const;

VBX

WLToolbar.Action = actionGetToolbarText(ByVal *nID* As Integer)

Notes

Sets the Text property if successful.

SetToolbarText API

Boolean SetToolbarText(*ButtonID***,***Text***)**

Sets the text for a toolbar or toolbar button.

Arguments

ButtonID	Button to set text for or WL_TOOLBARTEXT to set toolbar's text.
Text	Text to display for button or toolbar.

Return Value

TRUE if successful or FALSE if an error occurred.

See Also

GetToolbarText

C

BOOL WLSetToolbarText(HTOOLBAR *hToolbar*,UINT *nID*,LPCSTR *lpszText*);

C++

BOOL CWeblibToolbar::SetToolbarText(UINT *nID*,LPCSTR *lpszText*) const;

VBX

WLToolbar.Action = actionSetToolbarText(ByVal *nID* As Integer, ByVal *szText* As String)

GetToolbarFont API

Font GetToolbarFont(*Type***)**

Returns the font used to draw toolbar text or button text.

Arguments

Type	WL_BUTTONFONT or WL_TOOLBARFONT.

Return Value

Font for button or toolbar text if successful or NULL if an error occurred.

See Also

SetToolbarFont

C

HFONT WLGetToolbarFont(HTOOLBAR *hToolbar*,WORD *wType*);

C++

HFONT CWeblibToolbar::GetToolbarFont(WORD *wType*) const;

VBX

WLToolbar.Action = actionGetToolbarFont(ByVal *nType* As Integer)

Notes

Sets the Font property if successful.

SetToolbarFont API

Boolean SetToolbarFont(*Type***,***Font***)**

Sets the font used to draw toolbar text or button text.

Arguments

Type	WL_BUTTONFONT or WL_TOOLBARFONT.
Font	Handle of font to use for drawing text. The application is responsible for creating and deleting this font. Do not delete the font while the toolbar is using it (delete the toolbar before the deleting the font).

Return Value

TRUE if successful or FALSE if an error occurred.

See Also

GetToolbarFont

C

BOOL WLSetToolbarFont(HTOOLBAR *hToolbar*,WORD *wType*,HFONT *hFont*);

C++

BOOL CWeblibToolbar::SetToolbarFont(WORD *wType*,HFONT *hFont*) const;

VBX

WLToolbar.Action = actionSetToolbarFont(ByVal *nType* As Integer, ByVal *nFont* As Integer)

GetToolbarBkgnd API

Color GetToolbarBkgnd()

Returns the background color of the toolbar.

Return Value

RGB color of the toolbar background.

See Also

SetToolbarBkgnd

C

COLORREF WLGetToolbarBkgnd(HTOOLBAR *hToolbar*);

C++

COLORREF CWeblibToolbar::GetToolbarBkgnd() const;

VBX

WLToolbar.Action = actionGetToolbarBkgnd()

Notes

Sets the Color property if successful.

SetToolbarBkgnd API

Boolean SetToolbarBkgnd(*Color*)

Sets the background color of the toolbar.

Arguments

Color	New RGB color of toolbar background.

Return Value

TRUE if successful or FALSE if an error occurred.

See Also

GetToolbarBkgnd

C

BOOL WLSetToolbarBkgnd(HTOOLBAR *hToolbar*,COLORREF *crBackground*);

C++

BOOL CWeblibToolbar::SetToolbarBkgnd(COLORREF *crBackground*) const;

VBX

WLToolbar.Action = actionSetToolbarBkgnd(ByVal *lBkgndColor* As Long)

GetToolbarTextColor API

Color GetToolbarTextColor(*Type***)**

Returns the color used to draw toolbar text or button text.

Arguments

Type	WL_BUTTONTEXTCOLOR or WL_TOOLBARTEXTCOLOR.

Return Value

RGB color of button or toolbar text if successful or NULL if an error occurred.

See Also

SetToolbarTextColor

C

COLORREF WLGetToolbarTextColor(HTOOLBAR *hToolbar*,WORD *wType*);

C++

COLORREF CWeblibToolbar::GetToolbarTextColor(WORD *wType*) const;

VBX

WLToolbar.Action = actionGetToolbarTextColor(ByVal *nType* As Integer)

Notes

Sets the Color property if successful.

SetToolbarTextColor API

Boolean SetToolbarTextColor(*Type*,*Color*);

Sets the color used to draw toolbar text or button text.

Arguments

Type	WL_BUTTONTEXTCOLOR or WL_TOOLBARTEXTCOLOR.
Color	RBG color value for drawing text.

Return Value

TRUE if successful or FALSE if an error occurred.

See Also

GetToolbarTextColor

C

BOOL WLSetToolbarTextColor(HTOOLBAR *hToolbar*,WORD *wType*,COLORREF *crText*);

C++

BOOL CWeblibToolbar::SetToolbarTextColor(WORD *wType*,COLORREF *crText*) const;

VBX

WLToolbar.Action = actionSetToolbarTextColor(ByVal *nType* As Integer, ByVal *lTextColor* As Long)

AppendFormData API

FormData AppendFormData(*Name***,***Data***)**

Creates a url-encoded string of the form '*Name*=*Data*' and appends it to a data buffer, growing the buffer if needed. Blanks are converted to '+', strings are joined with '&' and unprintable and special characters are converted to '%XX' notation.

Arguments

Name	Name of field or variable.
Data	Value of field or variable

Return Value

Handle to data buffer if successful or NULL if an error occurred.

See Also

AccessFormData,GetFormDataLength

C/C++

HFORMDATA WLAppendFormData(HFORMDATA *hFormData*,LPCSTR *lpszName*,LPCSTR *lpszData*,UINT *cbData*);

Notes

To create a new form data buffer, pass *hFormData* as NULL. A handle to the new buffer is returned that may be used in subsequent calls to append data.

VBX

WLBrowser.Action = actionAppendFormData(ByVal *lpszName* As String,ByVal *lpszData* As String,ByVal *cbData* As Integer)

Notes

A new form data buffer is created in two cases: the first time this function is called for the control and on the first call to this function after actionAccessFormData().

GetFormDataLength API

Integer GetFormDataLength()

Returns the length of a form data buffer. The length includes the null terminator.

Return Value

Length of data buffer if successful or zero if an error occurred.

See Also

AppendFormData,AccessFormData

C/C++

UINT WLGetFormDataLength(HFORMDATA *hFormData*);

VBX

Not applicable.

AccessFormData API

Boolean AccessFormData(*ReturnBuffer*)

Copies a form data buffer to an application-supplied buffer, freeing the form data buffer.

Arguments

Return Buffer	Application buffer where form data is copied. Call the function GetFormDataLength() to determine how big this buffer should be.

Return Value

TRUE if successful or FALSE if an error occurred.

See Also

AppendFormData,GetFormDataLength

C/C++

BOOL WLAccessFormData(HFORMDATA *hFormData*,LPSTR *lpszRetBuf*,UINT *cbRetBuf*);

Notes

Once this function has been called, *hFormData* is no longer a valid buffer handle.

VBX

WLBrowser.Action = actionAccessFormData()

Notes

Sets the property FormData if successful.

HtmlParseFile API

Parse HtmlParseFile(*File***,***Options***)**

Parses the HTML document stored in *File* according to *Options*, creating a parse tree of HTML elements (e.g., tags and text).

Arguments

File	File that contains an HTML document.
Options	WL_KEEPCLOSINGTAG or WL_KEEPATTRIBUTETAG (or both).

Return Value

Handle to a parse tree if successful or NULL if an error occurred.

See Also

HtmlParseBuf,HtmlEndParse

C

HPARSE WLHtmlParseFile(LPCSTR *lpszFilename*,WORD *wOptions*);

C++

BOOL CWeblibHtml::ParseFile(LPCSTR *lpszFilename*,WORD *wOptions*);

VBX

WLParser.Action = actionHtmlParseFile(ByVal *lpszFile* As String,ByVal *wOptions* As Integer)

HtmlParseBuf API

Parse HtmlParseBuf(*Buf***,***Options***)**

Parses the HTML document stored in *Buf* according to *Options*, creating a parse tree of HTML elements (e.g., tags and text).

Arguments

Buf	Memory buffer that contains an HTML document.
Options	WL_KEEPCLOSINGTAG or WL_KEEPATTRIBUTETAG (or both).

Return Value

Handle to a parse tree if successful or NULL if an error occurred.

See Also

HtmlParseFile,HtmlEndParse

C

HPARSE WLHtmlParseBuf(LPCSTR *lpszBuf*,DWORD *cbBuf*,WORD *wOptions*);

Notes

The value of *cbBuf* indicates whether *lpszBuf* is a far or huge pointer. If *cbBuf* is greater than or equal to 64K, *lpszBuf* is assumed to be a huge pointer.

C++

BOOL CWeblibHtml::ParseBuf(LPCSTR *lpszBuf*,DWORD *cbBuf*,WORD *wOptions*);

Notes

The value of *cbBuf* indicates whether *lpszBuf* is a far or huge pointer. If *cbBuf* is greater than or equal to 64K, *lpszBuf* is assumed to be a huge pointer.

VBX

WLParser.Action = actionHtmlParseBuf(ByVal *lpszBuf* As String,ByVal *wOptions* As Integer)

Notes

The VBX version cannot be used to parse buffers larger than 64K.

HtmlEndParse API

Boolean HtmlEndParse()

Frees a parse tree.

Return Value

TRUE if successful or FALSE if an error occurred.

See Also

HtmlParseFile,HtmlParseBuf

C

BOOL WLHtmlEndParse(HPARSE *hParse*);

Notes

After this function is called, *hParse* is no longer a valid parse tree handle.

C++

BOOL CWeblibHtml::EndParse();

VBX

WLParser.Action = actionHtmlEndParse()

HtmlEnumParseTree API

Boolean HtmlEnumParseTree(*NotifyWindow***,***Message***)**

Enumerates a parse tree by sending the EnumParseTree event to *NotifyWindow* for each element in the tree.

Arguments

NotifyWindow	Handle of window to receive events.
Message	Notification message.

Return Value

TRUE if successful or FALSE if an error occurred.

See Also

EnumParseTree event

C

BOOL WLHtmlEnumParseTree(HPARSE *hParse*,HWND *hwndNotify*,UINT *nMsg*);

C++

BOOL CWeblibHtml::EnumParseTree(UINT *nEnumID*);

VBX

WLParser.Action = actionHtmlEnumParseTree(ByVal *nEnumID* As Integer)

HtmlGetChild API

Element HtmlGetChild(*Element***)**

Returns the child of a parse tree element.

Arguments

Element	Element to get child of.

Return Value

Child element if successful or NULL if *Element* has no children.

See Also

HtmlGetParent,HtmlGetSibling

C

HELEMENT WLHtmlGetChild(HPARSE *hParse*,HELEMENT *hElement*);

C++

HELEMENT CWeblibHtml::GetChild(HELEMENT *hElement*) const;

VBX

WLParser.Action = actionHtmlGetChild(ByVal *lElement* As Long)

Notes

Sets the Element property if successful.

HtmlGetParent API

Element HtmlGetParent(*Element***)**

Returns the parent of a parse tree element.

Arguments

Element	Element to get parent of.

Return Value

Parent element if successful or NULL if *Element* has no parent (is the root element).

See Also

HtmlGetChild,HtmlGetSibling

C

HELEMENT WLHtmlGetParent(HPARSE *hParse*,HELEMENT *hElement*);

C++

HELEMENT CWeblibHtml::GetParent(HELEMENT *hElement*) const;

VBX

WLParser.Action = actionHtmlGetParent(ByVal *lElement* As Long)

Notes

Sets the Element property if successful.

HtmlGetSibling API

Element HtmlGetSibling(*Element***,***Relationship***)**

Returns a sibling (element at the same level) of a parse tree element.

Arguments

Element	Element to get sibling of.
Relationship	Relationship of element to be returned to *Element*: WL_FIRSTELEM, WL_NEXTELEM, WL_PREVELEM or WL_LASTELEM.

Return Value

Sibling element if successful or NULL if *Element* has no such sibling.

See Also

HtmlGetParent,HtmlGetChild

C

HELEMENT WLHtmlGetSibling(HPARSE *hParse*,HELEMENT *hElement*,WORD *wRel*);

C++

HELEMENT CWeblibHtml::GetSibling(HELEMENT *hElement*,WORD *wRel*) const;

VBX

WLParser.Action = actionHtmlGetSibling(ByVal *lElement* As Long,ByVal *wRel* As Integer)

Notes

Sets the Element property if successful.

HtmlGetElementType API

Integer HtmlGetElementType(Element**)**

Returns the type of a parse tree element.

Arguments

Element	Element to get type of.

Return Value

WL_ROOT (root of tree), WL_TAG, WL_TEXT, WL_SPECIALCHAR (e.g., <) or WL_COMMENT

See Also

HtmlGetElementText

C

UINT WLHtmlGetElementType(HPARSE *hParse*,HELEMENT *hElement*);

C++

UINT CWeblibHtml::GetElementType(HELEMENT *hElement*) const;

VBX

WLParser.Action = actionHtmlGetElementType(ByVal *lElement* As Long)

Notes

Sets the Type property if successful.

HtmlGetElementText API

String HtmlGetElementText(*Element***)**

Returns the text associated with a parse tree element.

Arguments

Element	Element to get text for.

Return Value

Element text or NULL if element is not associated with any text (root element).

See Also

HtmlGetElementType

C

LPCSTR WLHtmlGetElementText(HPARSE *hParse*,HELEMENT *hElement*);

C++

LPCSTR CWeblibHtml::GetElementText(HELEMENT *hElement*) const;

VBX

WLParser.Action = actionHtmlGetElementText(ByVal *lElement* As Long)

Notes

Sets the Text property if successful.

HtmlGetTextAttr API

Long HtmlGetTextAttr(*Element***)**

Returns the text attribute bitmask associated with a text element.

Arguments

Element	Element to get text attributes for.

Return Value

Text attribute bitmask or zero if *Element* is not a text element.

See Also

HtmlGetElementType,HtmlGetElementText

C

DWORD WLHtmlGetTextAttr(HPARSE *hParse*,HELEMENT *hElement*);

C++

DWORD CWeblibHtml::GetTextAttr(HELEMENT *hElement*) const;

VBX

WLParser.Action = actionHtmlGetTextAttr(ByVal *lElement* As Long)

Notes

Sets the TextAttr property if successful.

HtmlGetTagName API

String HtmlGetTagName(*Element*)

Returns the name of a tag element.

Arguments

Element	Tag element to get name of.

Return Value

Tag name or NULL if *Element* is not a tag element.

See Also

HtmlGetElementType,HtmlGetElementText

C

LPCSTR WLHtmlGetTagName(HPARSE *hParse*,HELEMENT *hElement*);

C++

LPCSTR CWeblibHtml::GetTagName(HELEMENT *hElement*) const;

VBX

WLParser.Action = actionHtmlGetTagName(ByVal *lElement* As Long)

Notes

Sets the Text property if successful.

HtmlGetTagType API

Integer HtmlGetTagType(*Element***)**

Returns the type of a tag element.

Arguments

Element	Tag element to get type of.

Return Value

Tag type (e.g.,HTML_BODY) or zero if *Element* is not a tag element.

See Also

HtmlGetElementType,HtmlGetTagName

C

UINT WLHtmlGetTagType(HPARSE *hParse*,HELEMENT *hElement*);

C++

UINT CWeblibHtml::GetTagType(HELEMENT *hElement*) const;

VBX

WLParser.Action = actionHtmlGetTagType(ByVal *lElement* As Long)

Notes

Sets the Type property if successful.

HtmlGetTagAttr API

TagAttr HtmlGetTagAttr(*Element***,***TagAttr***,***Type***,***Attr***,***Value***)**

Gets the next attribute and its value from a tag element. For example, the tag <IMG SRC="my.gif" ALIGN=TOP> has three attributes:

IMG (stand-alone tag name)
SRC (value is "my.gif")
ALIGN (value is TOP)

HtmlGetTagAttr() may be used to enumerate each attribute in a tag and its value. To get the first attribute, pass *TagAttr* as NULL. A handle to the next attribute is returned and may be used to continue the enumeration.

Arguments

Element	Tag element to get attribute from.
TagAttr	Handle of tag attribute or NULL to get the first attribute.
Type	The attribute data type is returned here: WL_WORD, WL_NUMBER, WL_QUOTEDSTRING or WL_STANDALONE.
Attr	The name of the attribute is returned here.
Value	The value of the attribute is returned here.

Return Value

Tag type (e.g.,HTML_BODY) or zero if *Element* is not a tag element.

See Also

HtmlExtractTagAttr

C

HTAGATTR WLHtmlGetTagAttr(HPARSE *hParse*,HELEMENT *hElement*,HTAGATTR *hTagAttr*,WORD **pwType*,LPSTR *lpszAttr*,UINT *cbAttr*,LPSTR *lpszValue*,UINT *cbValue*);

C++

HTAGATTR CWeblibHtml::GetTagAttr(HELEMENT *hElement*,HTAGATTR *hTagAttr*,WORD **pwType*,LPSTR *lpszAttr*,UINT *cbAttr*,LPSTR *lpszValue*,UINT *cbValue*) const;

VBX

WLParser.Action = actionHtmlGetTagAttr(ByVal *lElement* As Long,ByVal *lTagAttr* As Long)

Notes

Sets the TagAttr,TagAttrType,TagAttrName and TagAttrValue properties if successful.

HtmlExtractTagAttr API

String HtmlExtractTagAttr(*Element***,***Attr***)**

Extracts the value of a tag attribute from a tag element.

Arguments

Element	Tag element to get attribute value from.
Attr	Name of tag attribute to get value of (e.g., ALIGN)

Return Value

Value of tag attribute or NULL if *Attr* cannot be found or an error occurred.

See Also

HtmlGetTagAttr

C

LPCSTR WLHtmlExtractTagAttr(HPARSE *hParse*,HELEMENT *hElement*,LPSTR *lpszAttr*);

C++

LPCSTR CWeblibHtml::ExtractTagAttr(HELEMENT *hElement*,LPSTR *lpszAttr*) const;

VBX

WLParser.Action = actionHtmlExtractTagAttr(ByVal *lElement* As Long,ByVal *lpszAttr* As String)

Notes

Sets the TagAttrValue property if successful.

HtmlFindText API

Element HtmlFindText(*Element*,*Text*)

Finds a text element that contains the string *Text*.

Arguments

Element	Search begins here and includes *Element*'s children.
Text	String to match.

Return Value

Element containing string or NULL if string was not found.

See Also

HtmlEnumFindText

C

HELEMENT WLHtmlFindText(HPARSE *hParse*,HELEMENT *hElement*,LPCSTR *lpszText*);

C++

HELEMENT CWeblibHtml::FindText(HELEMENT *hElement*,LPCSTR *lpszText*) const;

VBX

WLParser.Action = actionHtmlFindText(ByVal *lElement* As Long,ByVal *lpszText* As String)

Notes

Sets the Element property if successful.

HtmlFindSpecial API

Element HtmlFindSpecial(*Element***,***Special***)**

Finds a special character element that matches the string *Special*.

Arguments

Element	Search begins here and includes *Element*'s children.
Special	String to match.

Return Value

Element matching string or NULL if string was not found.

See Also

HtmlEnumFindSpecial

C

HELEMENT WLHtmlFindSpecial(HPARSE *hParse*,HELEMENT *hElement*,LPCSTR *lpszSpecial*);

C++

HELEMENT CWeblibHtml::FindSpecial(HELEMENT *hElement*,LPCSTR lpszSpecial) const;

VBX

WLParser.Action = actionHtmlFindSpecial(ByVal *lElement* As Long,ByVal *lpszSpecial* As String)

Notes

Sets the Element property if successful.

HtmlFindComment API

Element HtmlFindComment(*Element***,***Comment***)**

Finds a comment element that contains the string *Comment*.

Arguments

Element	Search begins here and includes *Element*'s children.
Comment	String to match.

Return Value

Element containing string or NULL if string was not found.

See Also

HtmlEnumFindComment

C

HELEMENT WLHtmlFindComment(HPARSE *hParse*,HELEMENT *hElement*,LPCSTR *lpszCommentText*);

C++

HELEMENT CWeblibHtml::FindComment(HELEMENT *hElement*,LPCSTR *lpszCommentText*) const;

VBX

WLParser.Action = actionHtmlFindComment(ByVal *lElement* As Long,ByVal *lpszComment* As String)

Notes

Sets the Element property if successful.

HtmlFindTagType API

Element HtmlFindTagType(*Element***,***Type***)**

Finds a tag element of a specific type.

Arguments

Element Search begins here and includes *Element*'s children.
Type Type of tag to find (e.g., HTML_BODY).

Return Value

Element of specified type or NULL if tag of this type was not found.

See Also

HtmlEnumFindTagType

C

HELEMENT WLHtmlFindTagType(HPARSE *hParse*,HELEMENT *hElement*,UINT *nType*);

C++

HELEMENT CWeblibHtml::FindTagType(HELEMENT *hElement*,UINT *nType*) const;

VBX

WLParser.Action = actionHtmlFindTagType(ByVal *lElement* As Long,ByVal *nType* As Integer)

Notes

Sets the Element property if successful.

HtmlFindTagName API

Element HtmlFindTagName(*Element***,***Tag***)**

Finds a tag element with a specific tag name.

Arguments

Element	Search begins here and includes *Element*'s children.
Tag	Tag name to find (e.g., "BODY").

Return Value

Element having the specified name or NULL if tag with this name was not found.

See Also

HtmlEnumFindTagName

C

HELEMENT WLHtmlFindTagName(HPARSE *hParse*,HELEMENT *hElement*,LPCSTR *lpszTag*);

C++

HELEMENT CWeblibHtml::FindTagName(HELEMENT *hElement*,LPCSTR *lpszTag*) const;

VBX

WLParser.Action = actionHtmlFindTagName(ByVal *lElement* As Long,ByVal *lpszTag* As String)

Notes

Sets the Element property if successful.

HtmlFindTagAttr API

Element HtmlFindTagAttr(*Element***,***Type***,***Attr***,***Value***)**

Finds a tag element having a certain type, tag attribute and value.

Arguments

Element	Search begins here and includes *Element*'s children.
Type	Type of tag to find (e.g., HTML_IMG).
Attr	Name of tag attribute to find (e.g., "ALIGN").
Value	Value of tag attribute to find (e.g., "TOP").

Return Value

Element having the specified name or NULL if tag with this name was not found.

See Also

HtmlEnumFindTagAttr

C

HELEMENT WLHtmlFindTagAttr(HPARSE *hParse*,HELEMENT *hElement*,UINT *nType*,LPCSTR *lpszAttr*,LPCSTR *lpszValue*);

C++

HELEMENT CWeblibHtml::FindTagAttr(HELEMENT *hElement*,UINT *nType*,LPCSTR *lpszAttr*,LPCSTR *lpszValue*) const;

VBX

WLParser.Action = actionHtmlFindTagAttr(ByVal *lElement* As Long,ByVal *nType* As Integer,ByVal *lpszAttr* As String,ByVal *lpszValue* As String)

Notes

Sets the Element property if successful.

HtmlEnumFindText API

Boolean HtmlEnumFindText(*Element*,*Text*,*NotifyWindow*,*Message***)**

Enumerates text elements that contains the string *Text*, sending the EnumFindText event to *NotifyWindow* for each match.

Arguments

Element	Search begins here and includes *Element*'s children.
Text	String to match.
NotifyWindow	Handle of window to receive events.
Message	Notification message.

Return Value

Returns TRUE if successful or FALSE if an error occurred.

See Also

HtmlFindText,EnumFindText event

C

BOOL WLHtmlEnumFindText(HPARSE *hParse*,HELEMENT *hElement*,LPCSTR *lpszText*,HWND *hwndNotify*,UINT *nMsg*);

C++

BOOL CWeblibHtml::EnumFindText(HELEMENT *hElement*,LPCSTR *lpszText*,UINT *nEnumID*);

VBX

WLParser.Action = actionHtmlEnumFindText(ByVal *lElement* As Long,ByVal *lpszText* As String,ByVal *nEnumID* As Integer)

HtmlEnumFindSpecial API

Boolean HtmlEnumFindSpecial(*Element***,***Special***,***NotifyWindow***,***Message***)**

Enumerates special character elements that match the string *Special*, sending the EnumFindSpecial event to *NotifyWindow* for each match.

Arguments

Element	Search begins here and includes *Element*'s children.
Special	String to match.
NotifyWindow	Handle of window to receive events.
Message	Notification message.

Return Value

Returns TRUE if successful or FALSE if an error occurred.

See Also

HtmlFindSpecial,EnumFindSpecial event

C

BOOL WLHtmlEnumFindSpecial(HPARSE *hParse*,HELEMENT *hElement*,LPCSTR *lpszSpecial*,HWND *hwndNotify*,UINT *nMsg*);

C++

BOOL CWeblibHtml::EnumFindSpecial(HELEMENT *hElement*,LPCSTR *lpszSpecial*,UINT *nEnumID*);

VBX

WLParser.Action = actionHtmlEnumFindSpecial(ByVal *lElement* As Long,ByVal *lpszSpecial* As String,ByVal *nEnumID* As Integer)

HtmlEnumFindComment API

Boolean HtmlEnumFindComment(*Element***,***Comment***,***NotifyWindow***,***Message***)**

Enumerates comment elements that contain the string *Comment*, sending the EnumFindComment event to *NotifyWindow* for each match.

Arguments

Element	Search begins here and includes *Element*'s children.
Comment	String to match.
NotifyWindow	Handle of window to receive events.
Message	Notification message.

Return Value

Returns TRUE if successful or FALSE if an error occurred.

See Also

HtmlFindComment,EnumFindComment event

C

BOOL WLHtmlEnumFindComment(HPARSE *hParse*,HELEMENT *hElement*,LPCSTR *lpszCommentText*,HWND *hwndNotify*,UINT *nMsg*);

C++

BOOL CWeblibHtml::EnumFindComment(HELEMENT *hElement*,LPCSTR *lpszCommentText*,UINT *nEnumID*);

VBX

WLParser.Action = actionHtmlEnumFindComment(ByVal *lElement* As Long,ByVal *lpszComment* As String,ByVal *nEnumID* As Integer)

HtmlEnumFindTagType API

Boolean HtmlEnumFindTagType(*Element***,***Type***,***NotifyWindow***,***Message***)**

Enumerates tag elements of a specific type, sending the EnumFindTagType event to *NotifyWindow* for each match.

Arguments

Element	Search begins here and includes *Element*'s children.
Type	Type of tag to find (e.g., HTML_BODY).
NotifyWindow	Handle of window to receive events.
Message	Notification message.

Return Value

Returns TRUE if successful or FALSE if an error occurred.

See Also

HtmlFindTagType,EnumFindTagType event

C

BOOL WLHtmlEnumFindTagType(HPARSE *hParse*,HELEMENT *hElement*,UINT *nType*,HWND *hwndNotify*,UINT *nMsg*);

C++

BOOL CWeblibHtml::EnumFindTagType(HELEMENT *hElement*,UINT *nType*,UINT *nEnumID*);

VBX

WLParser.Action = actionHtmlEnumFindTagType(ByVal *lElement* As Long,ByVal *nType* As Integer,ByVal *nEnumID* As Integer)

HtmlEnumFindTagName API

Boolean HtmlEnumFindTagName(*Element***,***Tag***,***NotifyWindow***,***Message***)**

Enumerates tag elements having a specific tag name, sending the EnumFindTagName event to *NotifyWindow* for each match.

Arguments

Element	Search begins here and includes *Element*'s children.
Tag	Tag name to find (e.g., "BODY").
NotifyWindow	Handle of window to receive events.
Message	Notification message.

Return Value

Returns TRUE if successful or FALSE if an error occurred.

See Also

HtmlFindTagName,EnumFindTagName event

C

BOOL WLHtmlEnumFindTagName(HPARSE *hParse*,HELEMENT *hElement*,LPCSTR *lpszTag*,HWND *hwndNotify*,UINT *nMsg*);

C++

BOOL CWeblibHtml::EnumFindTagName(HELEMENT *hElement*,LPCSTR *lpszTag*,UINT *nEnumID*);

VBX

WLParser.Action = actionHtmlEnumFindTagName(ByVal *lElement* As Long,ByVal *lpszTag* As String,ByVal *nEnumID* As Integer)

HtmlEnumFindTagAttr API

Boolean HtmlEnumFindTagAttr(*Element***,***Type***,***Attr***,***Value***,***NotifyWindow***,***Message***)**

Enumerates tag elements having of a certain type, tag attribute and value, sending the EnumFindTagAttr event to *NotifyWindow* for each match.

Arguments

Element	Search begins here and includes *Element*'s children.
Type	Type of tag to find (e.g., HTML_IMG).
Attr	Name of tag attribute to find (e.g., "ALIGN").
Value	Value of tag attribute to find (e.g., "TOP").
NotifyWindow	Handle of window to receive events.
Message	Notification message.

Return Value

Returns TRUE if successful or FALSE if an error occurred.

See Also

HtmlFindTagAttr,EnumFindTagAttr event

C

BOOL WLHtmlEnumFindTagAttr(HPARSE *hParse*,HELEMENT *hElement*,UINT *nType*,LPCSTR *lpszAttr*,LPCSTR *lpszValueText*,HWND *hwndNotify*,UINT *nMsg*);

C++

BOOL CWeblibHtml::EnumFindTagAttr(HELEMENT *hElement*,UINT *nType*,LPCSTR *lpszAttr*,LPCSTR *lpszValueText*,UINT *nEnumID*);

VBX

WLParser.Action = actionHtmlEnumFindTagAttr(ByVal *lElement* As Long,ByVal *nType* As Integer,ByVal *lpszAttr* As String,ByVal *lpszValue* As String,ByVal *nEnumID* As Integer)

ParseAbsoluteURL API

Boolean ParseAbsoluteURL(*URL***,***Protocol***,***Host***,***Port***,***Path***)**

Parses a URL into its components.

Arguments

URL	URL to parse (e.g., "http://www.xxx.com:80/dir/file").
Protocol	Protocol part of URL is copied here (e.g., "http").
Host	Host part of URL is copied here (e.g., "www.xxx.com").
Port	Port part of URL is copied here (e.g., 80); zero if none.
Path	Path part of URL is copied here (e.g., "dir/file").

Return Value

Returns TRUE if successful or FALSE if an error occurred.

C/C++

BOOL WLParseAbsoluteURL(LPCSTR *lpszURL*,LPSTR *lpszProtocol*,UINT *cbProtocol*,LPSTR *lpszHost*,UINT *cbHost*,UINT **pnPort*,LPSTR *lpszPath*,UINT *cbPath*);

VBX

WLParser.Action = actionParseAbsoluteURL(ByVal *lpszURL* As String)

Notes

Sets the properties Protocol, Host, Path and Port if successful.

Action Property

Action is a *Long* property which causes the VBX control to execute a function when set. This property should only be set using the return value of an action function (see the Visual Basic Overview). Accessing this property returns a constant that identifies the last action executed (e.g., ACTION_OPENURL).

Result Property

Result is a *Boolean* property that stores the status of the last action function executed. Make sure this property is TRUE before accessing any properties returned by an action function.

Text Property

Text is a *String* property that is returned by actionHtmlGetElementText() and actionHtmlGetTagName().

TagAttrName Property

TagAttrName is a *String* property that is returned by actionHtmlGetTagAttr().

TagAttrValue Property

TagAttrValue is a *String* property that is returned by actionHtmlGetTagAttr() and actionHtmlExtractTagAttr().

Protocol Property

Protocol is a *String* property that is returned by actionParseAbsoluteURL().

Host Property

Host is a *String* property that is returned by actionParseAbsoluteURL().

Path Property

Path is a *String* property that is returned by actionParseAbsoluteURL().

Element Property

Element is a *Long* property that is returned by actionHtmlGetChild(), actionHtmlGetParent() and several other action functions. This property is reset each time an action function is executed.

TextAttr Property

TextAttr is a *Long* property that is returned by actionHtmlGetTextAttr().

TagAttr Property

TagAttr is a *Long* property that is returned by actionHtmlGetTagAttr().

Type Property

Type is an *Integer* property that is returned by actionHtmlGetElementType() and actionHtmlGetTagType().

TagAttrType Property

TagAttrType is an *Integer* property that is returned by actionHtmlGetTagAttr().

Port Property

Port is an *Integer* property that is returned by actionParseAbsoluteURL().

Text Property

Text is a *String* property that is returned by actionGetToolbarText().

Font Property

Font is an *Integer* property that is returned by actionGetToolbarFont(). Although this property's data type is Integer, it really stores a font handle (HFONT).

Color Property

Color is an RGB *color* property that is returned by actionGetToolbarTextColor() and actionGetToolbarBkgnd().

AutoCreate Property

AutoCreate is a *Boolean* design-time property. If True, the toolbar is created at the same time as the VBX control.

ColorBkgnd Property

ColorBkgnd is an RGB *color* design-time property for the background color of the toolbar.

ColorToolbarText Property

ColorToolbarText is an RGB *color* design-time property for the toolbar text color.

ColorToolbarText Property

ColorButtonText is an RGB *color* design-time property for the button text color.

TextMenu Property

TextMenu is a *String* design-time property for the text that appears in the toolbar's popup menu.

TextToolbar Property

TextToolbar is a *String* design-time property for the toolbar text.

FontToolbar Property

FontToolbar is a *String* design-time property for the font used to draw the toolbar text.

FontButton Property

FontButton is a *String* design-time property for the font used to draw the button text.

Button Property

Button is an *enumerated* design-time property that specifies a button position in the toolbar.

ButtonID Property

ButtonID is an *Integer* design-time property for the button ID.

ButtonText Property

ButtonText is a *String* design-time property for the button text.

ButtonPic Property

ButtonPic is a *Picture* design-time property for the normal (up) bitmap.

ButtonPicSel Property

ButtonPicSel is a *Picture* design-time property for the selected (pressed) bitmap.

ButtonPicFocusProperty

ButtonPicFocus is a *Picture* design-time property for the focus bitmap.

ButtonPicDisabled Property

ButtonPicDisabled is a *Picture* design-time property for the disabled bitmap.

URL Property

URL is a *String* property that is returned by actionQueryURLFile(), actionGetWindowInfo() and actionParseAnchor().

Title Property

Title is a *String* property that is returned by actionGetWindowInfo().

Window Property

Window is a *Long* property that is returned by actionGetTransactionWindow() and actionActivateWindow().

WindowList Property

WindowList is a property array of *Longs* that is returned by actionListWindows(). The browser window IDs in the array are delimited by a zero.

MajorVersion Property

MajorVersion is an *Integer* property that is returned by actionGetVersion().

MinorVersion Property

MinorVersion is an *Integer* property that is returned by actionGetVersion().

FormData Property

FormData is a *String* property that is returned by actionAccessFormData().

Transaction Property

Transaction is a *Long* property that is returned by actionOpenURL(), actionPostFormData() and other action functions.

GenerateEvents Property

GenerateEvents is a *Boolean* property that controls event generation. Set this property to False to disable event generation for the next action function. After the action function is executed, this property is reset back to True.

OpenURL Event

OpenURL Event

Notifies application to open a URL.

Parameters

Window	Browser window to display document in.
Transaction	Transaction ID returned by API that sent event.
Flags	WL_NODOCUMENTCACHE, WL_NOIMAGECACHE or WL_BACKGROUNDMODE.
URL	URL to open.
MIME	MIME type of document to open (may be NULL or empty).
SaveFile	File to save URL in (may be NULL or empty).
FormData	Form data to post to URL (may be NULL or empty).

Return Value

Ignored.

See Also

RegisterProtocol

C

wParam contains WLN_OPENURL, extract parameters from *lParam* with **WLNGetWindow**(),**WLNGetTransaction**(),**WLNGetFlags**(),**WLNGetURL**(),**WLNGetMIMEType**(),**WLNGetSaveFile**() and **WLNGetFormData**().

C++

virtual void CWeblibBrowser::OnOpenURL(DWORD *dwTransaction*,DWORD *dwWindow*,DWORD *dwFlags*,LPCSTR *lpszURL*,LPCSTR *lpszMIME*,LPCSTR *lpszSaveFile*,LPCSTR *lpszFormData*);

VBX

OpenURL(*Window* As Long, *Txn* As Long, *Flags* As Long, *URL* As String, *MIMEType* As String,*SaveFile* As String, *FormData* As String)

BeginProgress Event

BeginProgress Event

Notifies application to initialize app-maintained progress indicator.

Parameters

Transaction	Transaction ID returned by API that sent event.
ProgressMsg	Progress string to display.

Return Value

Ignored.

See Also

OpenURL,SaveURL,PostFormData,SaveFormData,ShowFile

C

wParam contains WLN_BEGINPROGRESS, extract parameters from *lParam* with **WLNGetTransaction**() and WLNGetProgressString()

C++

virtual void CWeblibBrowser::OnBeginProgress(DWORD *dwTransaction*,LPCSTR *lpszProgress*);

VBX

BeginProgress(*Txn* As Long, *ProgressMessage* As String)

SetProgressRange Event

SetProgressRange Event

Notifies application of maximum progress value.

Parameters

Transaction	Transaction ID returned by API that sent event.
MaxProgress	Maximum progress value.

Return Value

Ignored.

See Also

OpenURL,SaveURL,PostFormData,SaveFormData,ShowFile

C

wParam contains WLN_SETPROGRESSRANGE, extract parameters from *lParam* with **WLNGetTransaction**() and **WLNGetProgressMaximum**().

C++

virtual void CWeblibBrowser::OnSetProgressRange(DWORD *dwTransaction*,DWORD *dwMaximum*);

VBX

SetProgressRange(*Txn* As Long, *MaxProgress* As Long)

MakingProgress Event

MakingProgress Event

Notifies application to update app-maintained progress indicator.

Parameters

Transaction	Transaction ID returned by API that sent event.
ProgressValue	New progress value (between zero and *ProgressMax*).
ProgressMsg	Progress string to display.

Return Value

Ignored.

See Also

OpenURL,SaveURL,PostFormData,SaveFormData,ShowFile

C

wParam contains WLN_MAKINGPROGRESS, extract parameters from *lParam* with **WLNGetTransaction**(),**WLNGetProgressValue**() and **WLNGetProgressString**().

C++

virtual void CWeblibBrowser::OnMakingProgress(DWORD *dwTransaction*,DWORD *dwProgress*,LPCSTR *lpszProgress*);

VBX

MakingProgress(*Txn* As Long, *Progress* As Long, *ProgressMessage* As String)

EndProgress Event

EndProgress Event

Notifies application to clear app-maintained progress indicator.

Parameters

Transaction Transaction ID returned by API that sent event.

Return Value

Ignored.

See Also

OpenURL,SaveURL,PostFormData,SaveFormData,ShowFile

C

wParam contains WLN_ENDPROGRESS, extract parameters from *lParam* with **WLNGetTransaction**().

C++

virtual void CWeblibBrowser::OnEndProgress(DWORD *dwTransaction*);

VBX

EndProgress(*Txn* As Long)

Finished Event

Finished Event

Notifies application that a transaction is complete.

Parameters

Transaction	Transaction ID returned by API that sent event.
Window	Browser window where document is displayed.

Return Value

Ignored.

See Also

OpenURL,SaveURL,PostFormData,SaveFormData,ShowFile

C

wParam contains WLN_FINISHED, extract parameters from *lParam* with **WLNGetTransaction**() and **WLNGetWindow**().

C++

virtual void CWeblibBrowser::OnFinished(DWORD *dwTransaction*,DWORD *dwWindow*);

VBX

Finished(*Txn* As Long, *Window* As Long)

Canceled Event

Canceled Event

Notifies application that a transaction was canceled.

Parameters

Transaction Transaction ID returned by API that sent event.

Return Value

Ignored.

See Also

OpenURL,SaveURL,PostFormData,SaveFormData,ShowFile,Cancel

C

wParam contains WLN_CANCELED, extract parameters from *lParam* with **WLNGetTransaction**().

C++

virtual void CWeblibBrowser::OnCanceled(DWORD *dwTransaction*);

VBX

Canceled(*Txn* As Long)

URLEcho Event

URLEcho Event

Notifies application that a URL was loaded.

Parameters

Window	Browser window where document is displayed.
URL	URL that was loaded.
MIME	MIME type of document that was loaded.
Referrer	URL where load was launched (previous URL).

Return Value

Ignored.

See Also

RegisterURLEcho

C

wParam contains WLN_URLECHO, extract parameters from *lParam* with **WLNGetTransaction**(),**WLNGetURL**(),**WLNGetMIMEType**() and **WLNGetReferrer**().

C++

virtual void CWeblibBrowser::OnURLEcho(DWORD *dwWindow*,LPCSTR *lpszURL*,LPCSTR *lpszMIME*,LPCSTR *lpszReferrer*);

VBX

URLEcho(*Window* As Long, *URL* As String, *MIMEType* As String, *Referrer* As String)

WindowChange Event

WindowChange Event

Notifies application that the state, position or size of browser window has changed.

Parameters

Window	Browser window that changed.
Flags	WLF_MOVESIZE, WLF_MAXIMIZED, WLF_NORMALIZED, WLF_MINIMIZED,WLF_CLOSED or WLF_EXITING.
X	New X coordinate of window.
Y	New Y coordinate of window.
Width	New width of window.
Height	New height of window.

Return Value

Ignored.

See Also

RegisterWindowChange

C

wParam contains WLN_WINDOWCHANGE, extract parameters from *lParam* with **WLNGetWindow**(),**WLNGetFlags**(),**WLNGetX**(),**WLNGetY**(),**WLNGetWidth**() and **WLNGetHeight**().

C++

virtual void CWeblibBrowser::OnWindowChange(DWORD *dwWindow*,DWORD *dwFlags*,DWORD *dwX*,DWORD *dwY*,DWORD *dwWidth*,DWORD *dwHeight*);

VBX

WindowChange(*Window* As Long, *Flags* As Long, *XPos* As Long, *YPos* As Long,*XSize* As Long, *YSize* As Long)

QueryViewer Event

QueryViewer Event

Notifies application to provide a filename to store a document for viewing.

Parameters

URL	URL of document to view.
MIME	MIME type of document to view.

Return Value

Ignored.

See Also

RegisterViewer,SetFileName

C

wParam contains WLN_QUERYVIEWER, extract parameters from *lParam* with **WLNGetURL**() and **WLNGetMIMEType**(). The application must set a filename with WLNSetFileName().

C++

virtual void CWeblibBrowser::OnQueryViewer(LPCSTR *lpszURL*,LPCSTR *lpszMIME*,LPSTR *lpszFile*,UINT *cbFile*);

Notes

A default filename is provided in *lpszFile* which the app may change.

VBX

QueryViewer(*URL* As String, *MIMEType* As String, *File* As String)

Notes

A default filename is provided in *File* which the app may change.

ViewDocFile Event

ViewDocFile Event

Notifies application to view a document.

Parameters

URL	URL of document.
MIME	MIME type of document.
File	File containing document.
Window	Browser window that initiated event.

Return Value

Ignored.

See Also

RegisterViewer

C

wParam contains WLN_VIEWDOCFILE, extract parameters from *lParam* with **WLNGetURL**(),**WLNGetMIMEType**(),**WLNGetFileName**() and **WLNGetWindow**().

C++

virtual void CWeblibBrowser::OnViewDocFile(DWORD *dwWindow*,LPCSTR *lpszURL*,LPCSTR *lpszMIME*,LPCSTR *lpszFile*);

VBX

ViewDocFile(*URL* As String, *MIMEType* As String, *File* As String, *Window* As Long)

ButtonClicked Event

ButtonClicked Event

Notifies application that a toolbar button was clicked.

Parameters

ButtonID Button that was clicked.

Return Value

Ignored.

See Also

CreateToolbar

C

wParam contains WLN_BUTTONCLICKED, extract parameters from *lParam* with **WLNGetButtonID**().

C++

virtual void CWeblibToolbar::OnButtonClicked(UINT *nButtonID*);

VBX

ButtonClicked(*ButtonID* As Integer)

EnumParseTree Event

EnumParseTree Event

Notifies application of an element in a parse tree enumeration.

Parameters

Type	Type of element (e.g., WL_TAG).
Element	Handle of HTML element.

Return Value

TRUE to continue enumeration or FALSE to stop enumeration.

See Also

HtmlEnumParseTree

C

wParam contains Type, *lParam* contains Element.

C++

virtual BOOL CWeblibHtml::OnParseTree(WORD *wType*,HELEMENT *hElement*,UINT *nEnumID*);

VBX

EnumParseTree(*EnumID* as Integer, *ElementType* As Integer, *Element* As Long, *ContinueEnum* As Integer)

EnumFindText Event

EnumFindText Event

Notifies application of an element in a 'find text' enumeration.

Parameters

Type	Type of element (e.g., WL_TAG).
Element	Handle of HTML element.

Return Value

TRUE to continue enumeration or FALSE to stop enumeration.

See Also

HtmlEnumFindText

C

wParam contains Type, *lParam* contains Element.

C++

virtual BOOL CWeblibHtml::OnFindText(WORD *wType*,HELEMENT *hElement*,UINT *nEnumID*);

VBX

EnumFindText(*EnumID* as Integer, *ElementType* As Integer, *Element* As Long, *ContinueEnum* As Integer)

EnumFindSpecial Event

EnumFindSpecial Event

Notifies application of an element in a 'find special character' enumeration.

Parameters

Type Type of element (e.g., WL_TAG).
Element Handle of HTML element.

Return Value

TRUE to continue enumeration or FALSE to stop enumeration.

See Also

HtmlEnumFindSpecial

C

wParam contains Type, *lParam* contains Element.

C++

virtual BOOL CWeblibHtml::OnFindSpecial(WORD *wType*,HELEMENT *hElement*,UINT *nEnumID*);

VBX

EnumFindSpecial(*EnumID* as Integer, *ElementType* As Integer, *Element* As Long, *ContinueEnum* As Integer)

EnumFindComment Event

EnumFindComment Event

Notifies application of an element in a 'find comment' enumeration.

Parameters

Type	Type of element (e.g., WL_TAG).
Element	Handle of HTML element.

Return Value

TRUE to continue enumeration or FALSE to stop enumeration.

See Also

HtmlEnumFindComment

C

wParam contains Type, *lParam* contains Element.

C++

virtual BOOL CWeblibHtml::OnFindComment(WORD *wType*,HELEMENT *hElement*,UINT *nEnumID*);

VBX

EnumFindComment(*EnumID* as Integer, *ElementType* As Integer, *Element* As Long, *ContinueEnum* As Integer)

EnumFindTagType Event

EnumFindTagType Event

Notifies application of an element in a 'find tag type' enumeration.

Parameters

Type Type of element (e.g., WL_TAG).
Element Handle of HTML element.

Return Value

TRUE to continue enumeration or FALSE to stop enumeration.

See Also

HtmlEnumFindTagType

C

wParam contains Type, *lParam* contains Element.

C++

virtual BOOL CWeblibHtml::OnFindTagType(WORD *wType*,HELEMENT *hElement*,UINT *nEnumID*);

VBX

EnumFindTagType(*EnumID* as Integer, *ElementType* As Integer, *Element* As Long, *ContinueEnum* As Integer)

EnumFindTagName Event

EnumFindTagName Event

Notifies application of an element in a 'find tag name' enumeration.

Parameters

Type	Type of element (e.g., WL_TAG).
Element	Handle of HTML element.

Return Value

TRUE to continue enumeration or FALSE to stop enumeration.

See Also

HtmlEnumFindTagName

C

wParam contains Type, *lParam* contains Element.

C++

virtual BOOL CWeblibHtml::OnFindTagName(WORD *wType*,HELEMENT*hElement*,UINT *nEnumID*);

VBX

EnumFindTagName(*EnumID* as Integer, *ElementType* As Integer, *Element* As Long, *ContinueEnum* As Integer)

BrowserExit Event

BrowserExit Event

Notifies application that the Web browser has terminated.

Parameters

None.

Return Value

None.

See Also

SetDefaultNotify

C/C++

wParam contains WLN_BROWSEREXIT, *lParam* is not used.

VBX

Not currently supported.

BrowserStart Event

BrowserStart Event

Notifies application that the Web browser has started running.

Parameters

None.

Return Value

None.

See Also

SetDefaultNotify

C/C++

wParam contains WLN_BROWSERSTART, *lParam* is not used.

VBX

Not currently supported.

EnumFindTagAttr Event

EnumFindTagAttr Event

Notifies application of an element in a 'find tag attribute' enumeration.

Parameters

Type Type of element (e.g., WL_TAG).
Element Handle of HTML element.

Return Value

TRUE to continue enumeration or FALSE to stop enumeration.

See Also

HtmlEnumFindTagAttr

C

wParam contains Type, *lParam* contains Element.

C++

virtual BOOL CWeblibHtml::OnFindTagAttr(WORD *wType*,HELEMENT *hElement*,UINT *nEnumID*);

VBX

EnumFindTagAttr(*EnumID* As Integer, *ElementType* As Integer, *Element* As Long, *ContinueEnum* As Integer)

WebLib Software License Agreement

Please see the file **LIC.TXT** that is included with the software.

API Reference

Housekeeping APIs

Startup
Cleanup
SetDefaultNotify

Web Browser APIs

ConnectBrowser
DisconnectBrowser
GetWindowInfo
ListWindows
ActivateWindow
CloseWindow
SetWindowPos
ShowWindow
ShowFile
OpenURL
SaveURL
PostFormData
SaveFormData
Cancel
RegisterProtocol
UnregisterProtocol
RegisterURLEcho
UnregisterURLEcho
RegisterViewer
UnregisterViewer
RegisterWindowChange
UnregisterWindowChange
ParseAnchor
GetVersion
QueryURLFile
SetNotifyMethod
GetTransactionWindow
SetFileName

Toolbar APIs

CreateToolbar
DeleteToolbar
SetActiveToolbar
IsToolbarActive
AddToolbarButton
RemoveToolbarButton
IsToolbarButtonVisible
ShowToolbarButton
IsToolbarButtonEnabled
EnableToolbarButton
GetToolbarText
SetToolbarText
GetToolbarFont
SetToolbarFont
GetToolbarBkgnd
SetToolbarBkgnd
GetToolbarTextColor
SetToolbarTextColor

HTML Parsing APIs

HtmlParseFile
HtmlParseBuf
HtmlEndParse
HtmlEnumParseTree
HtmlGetChild
HtmlGetParent
HtmlGetSibling
HtmlGetElementType
HtmlGetElementText
HtmlGetTextAttr
HtmlGetTagName
HtmlGetTagType
HtmlGetTagAttr
HtmlExtractTagAttr
HtmlFindText
HtmlFindSpecial
HtmlFindComment
HtmlFindTagType
HtmlFindTagName
HtmlFindTagAttr
HtmlEnumFindText
HtmlEnumFindSpecial
HtmlEnumFindComment
HtmlEnumFindTagType
HtmlEnumFindTagName
HtmlEnumFindTagAttr

Utility APIs

AppendFormData
GetFormDataLength
AccessFormData
ParseAbsoluteURL

VBX Properties

Each VBX control contains the Action and Result properties. The properties that are specific to a control are listed below.

WLBrowser Properties

URL	Title	Window
WindowList	MajorVersion	MinorVersion
FormData	Transaction	GenerateEvents

WLToolbar Properties

Text	Font	Color
AutoCreate	ColorBkgnd	ColorToolbarText
ColorButtonText	TextMenu	TextToolbar
FontToolbar	FontButton	Button
ButtonID	ButtonText	ButtonPic
ButtonPicSel	ButtonPicFocus	ButtonPicDisabled

WLHtmlParser Properties

Text	TagAttrName	TagAttrValue
Protocol	Host	Path
Element	TextAttr	TagAttr
Type	TagAttrType	Port

Events

Global Events

BrowserExit
BrowserStart

Web Browser Events

BeginProgress Event
SetProgressRange Event
MakingProgress Event
EndProgress Event
Finished Event
Canceled Event
OpenURL Event
URLEcho Event
WindowChange Event
QueryViewer Event
ViewDocFile Event

Toolbar Events

ButtonClicked Event

HTML Parsing Events

EnumParseTree Event
EnumFindText Event
EnumFindSpecial Event
EnumFindComment Event
EnumFindTagType Event
EnumFindTagName Event
EnumFindTagAttr Event

Parse Tree Example

Here is a very simple HTML document:

```
<html>
This is <b>BOLD</b> text. How does it look?
<P>
OK!
<P>
<IMG SRC="http://host/file">
</html>
```

The parse tree created by HtmlParseFile() (or HtmlParseBuf()) for this example document follows:

This tree has three levels: the root is at the top, under the root is the <*html*> container tag and under <*html*> we find the leaves of the tree. Like the example document, the tree is very simple and does not contain any comment elements (WL_COMMENT) or special character elements (WL_SPECIALCHAR).

The example tree was parsed with both the WL_KEEPCLOSINGTAG and WL_KEEPATTRIBUTETAG options. Omitting both of these options would drop three elements from the tree: the two HTML_BOLD tags (i.e., <*b*> and </*b*>) and the closing HTML_HTML tag (i.e., </*html*>).

Lastly, note that the HTML_IMG tag (i.e., <*IMG SRC="http://host/file"*>) has two tag attributes which are not explicity shown in the parse tree diagram. The first attribute is the *IMG*tag name and is of type WL_STANDALONE. The second tag attribute is *SRC* which is of type WL_QUOTEDSTRING and has the value *http://host/file*.