ActiveX Programming

http://kickme.to/tiger/

ActiveX Programming Unleashed

Chapter 1

An Overview of ActiveX

by Weiying Chen

Microsoft has unveiled an extensive new solution technology for the Internet called *ActiveX*. Microsoft ActiveX is a broad and powerful abstraction for Microsoft Internet Solutions. Content providers and Internet application developers now have a robust and extensible frameworks that enables them to develop a new generation of Internet applications.

Microsoft is aggressively adding features to the Win32 *Application Programming Interfaces* (*API*s) that will let developers "Internet-enable" their applications. These new features are based on *Object Linking and Embedding* (*OLE*) technology so that developers who have made investments into Win32 and OLE applications can leverage their investments.

ActiveX exposes a set of APIs that enables developing a new generation of client/server applications for the Internet. ActiveX has interfaces to integrate almost every media technology within an application. It provides extensive support for animation, 3D virtual reality, real-time audio, and real-time video.

ActiveX gives developers an open framework for building innovative applications for the Internet. ActiveX technologies form a robust framework for creating interactive content using reusable components, scripts, and existing applications. Specifically, ActiveX technologies enable content providers and application developers to create powerful and dynamic Web content and Web server extensions quite easily. This feat is achieved by using ActiveX controls, Active client and server side scripts, and the Active document interfaces and ISAPI (Internet Server Application Programming Interface).

COM: The Fundamental "Object Model" for ActiveX and OLE

COM (Component Object Model) is the technical cornerstone for the ActiveX technology; it defines how objects expose themselves for use within other objects and how objects can communicate between processes and across a network. You can easily integrate COM objects for use in many languages, such as Java, Basic, and C++. COM *objects* are reusable binary components.

The following concepts are fundamental to COM:

- Interface: The mechanism through which an object exposes itself.
- IUnknown Interface: The interface on which all others are based. It implements the reference-counting and interface-querying mechanisms required for COM objects.
- Reference Counting: The technique by which an object keeps track of its reference instance count. The instance of the object class should be deleted when there is no reference to this instance.
- QueryInterface Method: It is called with the Interface ID (IID) to which the caller wants a pointer. can be generated by Guidgen.exe by choosing DEFINE_GUID(...) format. QueryInterface enables navigation to other interfaces exposed by the object.
- IClassFactory Interface: This interface must be implemented for every object class. It provides functionality to create an instance of the object class with CLSID and locks the object server in memory to allow creation of objects more quickly.
- Marshaling: The mechanism that enables objects to be used across process and network boundaries, allowing interface parameters for location independence by packing and sending them across the process boundary. Developers have to create proxy/stub dll for the custom interfaces if exist. The custom interface has to be registered in the system registry.
- Aggregation: COM object supports an interface by including another object that supports that interface. The containing object creates the contained object as part of its own creation. The result is that the containing object exports the interface for the contained object by not implementing that interface.
- Multiple Inheritance: A derived class may inherit from multiple interfaces.

ActiveX Object Model

There are two primary pieces to the ActiveX Object Model: the Microsoft HyperText Markup Language (HTML) Viewer component (MSHTML.dll) object and the Web Browser Control (shdocvw.dll). Both are in-process (DLL-based) COM objects.classes.

All interfaces defined in the ActiveX Object Model are "dual" interfaces. A "dual" interface means that the objects inherit from IDispatch and IUnknown. They can be used by client application at "early-bind" via Vtable and at "late bind" via OLE automation controller by using IDispatch::GetIdsOfNames and IDispatch::Invoke.vtable).

MSHTML is the HTML viewer part of Microsoft Internet Explorer 3.0. It is an in-process COM server and a Document Object. It can be hosted in OLE Document Object containers.

MSHTML implements the OLE Automation object model described in the HTML Scripting Object Model. With this object model, you can develop rich multimedia HTML content. VBScript running inline in the HTML and Visual Basic 4.0 running external to the HTML can use the object model.

The Web browser control object is an in-process COM Server. It also serves as a Document Object container that can host any Document Objects, including MSHTML, with the added benefit of fully supporting hyperlinking to any document type.

The Web browser control is also an OLE control. The IWebBrowser interface is the primary interface exposed by the Web Browser Control.

The Web browser control is the core of what customers see as "the Internet Explorer 3.0 product.". Internet Explorer 3.0 also provides a frame to host this control. Internet Explorer 3.0 supports the following HTML 3.x0 extensions:

- Frame: Creates permanent panes for displaying information, supporting floating frames or borderless frames..
- NOFRAMES: Content that can be viewed by browsers not supporting frames
- OBJECT: Inserts an OLE control
- TABLE: Fully compliant with HTML 3.x0 tables with cell shading and text wrapping.
- StyleSheet: font size, intra-line space, margin, highlighting and other features related with styles can be specified in the HTML by the user.
- In-Line sound and video

ActiveX Controls

ActiveX Control (formerly known as OLE control) has a broader definition. It refers to any COM objects. For instance, the following objects are all considered an ActiveX control.

- Objects that expose a custom interface and the IUnknown interface
- OLE automation servers that expose the IDispatch/Dual interfaces
- Existing OLE controls (OCX)
- OLE objects that make use of monikers
- Java Applet with the support of COM

Note

A moniker acts as a name that uniquely identifies a COM object. It is a perfect programming abstraction for Uniform resource locator(URL).

ActiveX Control used inside scripting languages make this binary reusable components reused in the Internet world. Almost any type of media wrapped into an ActiveX control can be seamlessly integrated into your Web page. Sound, video, animation, or even credit-card approvals controls can be used within your Web page.

ActiveX Scripting

ActiveX Scripting is the interface for script engines and script hosts. Following this interface specification, the script vendors can use their custom script engine in any script host, such as IE 3.0. What's more, with its infrastructure, developers can choose any script language they prefer. A *script* is some executable block, such as a DOS batch file, Perl script or an EXE file.

ActiveX Scripting components can be grouped into two major categories: an ActiveX Scripting Engine and an ActiveX Scripting host. A *host* creates a script engine so that scripts can run against the host.

Here are some examples of ActiveX Scripting hosts:

- Microsoft Internet Explorer
- Internet Authoring tools
- Shell

The ActiveX Scripting Engine is an OLE COM object that supports the IOLEScript interfaces, with at least one of the IPersist interfaces and an optional IOleScriptParse interface.

ActiveX Scripting Engines can be developed for any language, such as

- Microsoft Visual Basic Script Edition (VBScript)
- JavaScript
- Perl

Microsoft ActiveX Scripting Languages products, such as Visual Basic Script and Java Script, can be used to "glue" together the functionality exposed in ActiveX Controls to create rich Web-based

applications.

Active Documents

Active Documents are based on the OLE Document Objects (DocObjects, for short). The DocObjects technology is a set of extensions to OLE Compound Document technology. It is the core technology that makes Microsoft Office Binder work. Active Document also refers to any document that contains ActiveX controls, a Java Applet, or a Document Object.

One other obvious application for this technology is "Internet browsers." You can open richly formatted documents, such as Microsoft Word and Excel spreadsheets, directly in the browser.

Figure 1.1 shows how seamlessly the Word document placed in IE 3.0. The word *toolbar* is added to the browser, enabling you to work on the document while cruising the Internet.

Figure 1.1. Active document.

The following describes the general criteria for the document object container and the document object.

A Document Object container must implement the following objects and interfaces:

- Document Site objects with IOleClientSite, IAdviseSink, and IOleDocumentSite interface exposed.
- View Site objects with IOleInPlaceSite and IContinueCallBack interface exposed.
- Frame Objects with IOleIPlaceFrame and IOleCommandTarget interface exposed
- IOleDocumentSite to support the DocObjects
- IOleCommandTarget on the frame object
- IOleClientSite and IAdvisesink for "site" object.
- IPersistStorage to handle object storage
- IOleInPlaceSite to support in-place activation of embedded object
- IOleInPlaceFrame for container's frame object

Similarly, Document Objects must implement the following objects and interfaces:

Objects

- Document Object with IDataObject, IPersistStorage, IPersistFile, and IOleDocument interface exposed.
- View Object with IOleInPlaceObject, IOleInPlaceActiveObject, IOleDocumentView, IPrint, and IOleCommandTarget interface exposed.

Containers

- IPersistStorage for storage mechanism
- IOleInPlaceObject and IOleInPlaceActiveObject for in-place activation extension of OLE

documents.

- IPersistFile, IOleObject, and IDataObect to support the basic embedding features of OLE documents.
- IOleDocument, IOleDocumentView, IOleCommandTarget, and IPrint to support the Document Objects extensions interface.

For more information on the Document Object, please refer to Chapter 19, "OLE Document Objects," in this book.

ActiveX Server and Server-Side Scripting Framework

The *ActiveX Server Framework*, another component of ActiveX technologies, is based on the Microsoft Internet Information Server (IIS). IIS is built on the Windows NT Advanced Server 3.51 or greater. This framework enables developers to take advantage of the powerful Microsoft Back Office family of products, such as Microsoft SQL, SNA, Management, and Exchange Server.

Server support consists of ActiveX Server-side scripting and the usage of Aside, Batch, or JavaScript.

The *Common Gateway Interface (CGI)* is also supported under the ActiveX Server Framework. Common Gateway Interface is a protocol used to communicate between your HTML forms and your program so that your program can extract the information from the form. A lot of languages can be used to write your program, as long as the language has the capability to read the STDIN, write to the STDOUT, and read the environment variables.

An HTTP server responds to a CGI execution request from a client browser by creating a new process, reading the input from the form through the environment variable, doing some processing with the form data and write the HTML response to the STDOUT.

The server creates one process for each CGI request received. However, creating a process for every request is time consuming and takes a lot of server resources. Using too many server resources can starve the server itself.

One way to avoid this is to convert the current CGI executable file into a Dynamic Link Library(DLL) so that the DLL can be loaded into the same address space as the server. The server can load the DLL the first time it gets a request. The DLL then stays in memory, ready to service other requests until the server decides it is no longer needed. There are two types of DLLs that can be created for this purpose: one is the Internet Server application (ISA), the other is the ISAPI filter.

An ISA(Internet server application) is also known as the Internet Server Extension. There are two entry points for this DLL, GetExtensionVersion and HttpExtensionProc.

HTTP Server first calls the ISA at the entry point of GetExtensionVersion to retrieve the version number of the ISAPI specification on which the extension is based. For every client request, the

HttpExtensionProc entry point is called. Interaction between an HTTP server and an ISA is done through extension control blocks (ECBs). The ISA must be multithread-safe because multiple requests can be received simultaneously.

An ISAPI filter is a replaceable DLL that sits on the HTTP server to filter data traveling between web browser and HTTL server. Figure 1.2 shows the relationship between the HTTP server, ISA, and ISAPI filter.

Figure 1.2. HTTP server, ISA, and ISAPI filter.

ISAPI filter has the same entry point as the ISA. When the filter is loaded, it tells the server what sort of notifications it will accept. After that, whenever a selected event occurs, the filter is called and is given the opportunity to process the event.

ISAPI filters are very useful to provide the following functions:

- Authentication
- Compression
- Encryption
- Logging of HTTP requests

You can also install Multiple filters can be installed on the server. The notification order is based on the priority specified by the filter, then the load order in the registry, the registry key for filter is under \HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\W3SVC\parameters\Filter DLLs\. After you install the filter, IIS must be restarted.

To ease the development of the server-side application, Microsoft also provides an Internet Personalization System (IPS) on top of the IIS. It provides a server side VBScript and a set of server components and a system installed with IIS on a web server computer to make the server-side scripting available. With this system installed, you can integrate Visual Basic Script commands and expressions into the HTML files, and calls out to server side OLE automation objects. When you point your browser to a script file, this system will process the script and return HTML to the browser. This allows developers to create more elaborate content with less effort. Server-side scripting will make the server-side development a piece of cake.

Active Animation and Movies

Active Animation (formally known as Active VRML; VRML stands for Virtual Reality Modeling Language) is an advanced markup language and a viewer for 3D multimedia. The Active Animation viewer is an ActiveX control and can be used inside client side VBScript in Microsoft Internet Explorer 3.0. Web pages that use Active VRML can include interactive 2D and 3D animation, accompanied by synchronized sounds. These effects can also be triggered by events from VBScript in IE 3.0 and from other scripting languages such as JavaScript.

Active Animation synchronizes all media interaction without developers needing to write low-level code.

It supports the operation and media types such as 3D geometry, images, sound, montages, 2D and 3D points, and vectors, 2D and 3D transforms, colors, numbers, text, strings, characters, and so forth.

- Animation: The following is a list of the operations and media types supported by Active Animation:
- Sound: Supports for importing, mixing, and rendering 3D models of sound waves.
- Images: Support infinite resolutions, 2D transformation, overlaying, and rendering images from 3D models.
- 3D geometry: Supports texture mapping of animated images combined with control of color and opacity, aggregation, and transformations.
- Advanced 2D and 3D coordinate systems and transformations: Supports many advanced manipulations of vector/vector, point/vector, and scalar/vector. Also supports construction and destruction of rectangular or polar/spherical coordinates. Translate, scale, rotate, shear, inversion, identity, composition and matrix-based constructions.
- Montages: Supports multi-layered cel animation and 2 ½-D images.

Note

- The word *cel* is synonymous with a frame of animation.
- Text: Supports formatted text, its colors and its fonts families including optional bold and italics.
- Colors: Construction and destruction in RGB and HSL colors.

The Active Movie is based on the Microsoft Active Movie Streaming Format (.ASF), a new data-independent format for storing and transmitting multimedia content across the Internet. Because .ASF files can be streamed, you can begin playback of these files immediately. Active Movie is an ActiveX control and can be used inside client-side VBScript or JavaScript.

.asf is an open and extendible data-independent format. With this format, you can combine and store different data objects such as audio objects, video objects, URLs, and HTML pages into a single synchronized multimedia stream. This encapsulation feature enables popular media types and formats, such as MPEG, .avi, .wav, and Apple QuickTime, to be synchronized and stored efficiently on a variety of servers.

.asf data is also network independent and can be transmitted over different protocols and networks, including TCP/IP, UDP, RTP, IPX/SPX, and ATM.

.asf also provides efficient packaging for different network transports, supports multiple-bit rates, error correction, and other multimedia-content storage and transmissions.

You can efficiently play back .asf content by using Active Movie, Microsoft's next generation, cross-platform video technology for the desktop.

Why Use ActiveX?

With ActiveX, you can make the most of Internet resources with less effort. ActiveX Controls and

Scripting give you the infrastructure needed to add language- and tool-independent extensions to Web pages. Using ActiveX Controls lets developers take advantage of existing OLE development tools and the investment they have already made in OLE. ActiveX Scripting allows you to drop any scripting engine into IE 3.0, enabling developers to add behavior to Web pages in whatever scripting language they prefer.

ActiveX has also greatly improved extending the HTTP and FTP protocols. The ActiveX IBindXXX interfaces encapsulate a new protocol that supports the concept of binding to a URL dynamically from within your application. An application binds to a URL moniker, which then communicates through the appropriate protocol to activate the OLE object. This abstraction allows newly developed protocols to integrate into your existing objects, independent of your object design or expressed functionality.

Using the Internet Extensions for the Win32 API (WinINet) makes it easy for developers to add Internet access to their applications. WinINet abstracts the TCP/IP protocol-specific details and gives developers a simplified programming interface to access the internet instead of worrying about the WinSocket details. This API includes HTTP, FTP and Gopher access.

As with most systems, server efficiency and resource use becomes a concern when designing multi-user server applications for the Internet. The Internet Information Server offers a high-performance, secure, and extendible framework. An ISAPI Internet Server Applications (ISA) is a dynamic link library that loads into the same address space as the HTTP Server component versus CGI creates a separate process for every work request. Each new process in the CGI model requires more server resources. The advantage of developing an ISA rather than a CGI is high-level performance that requires considerably fewer resources, which frees up resources that can then be used by the server.

The Internet Database Connector allows ODBC database access through an HTTP request.

ISAPI filters can be used to enhance the Microsoft Internet Information Server with custom features, such as enhanced logging of HTTP requests, custom encryption, aliases for URLs, compression schemes, or new authentication methods. The filter applications sit between the client network connection to the HTTP server.

IIS has a few built-in ISAPI DLL. One of them is Httpodbc.dll which is called the Internet Database Connector. The Internet Database Connector allows ODBC database access through an HTTP request. Developers can use this feature to create Web pages with information from the database so that they can retrieve, insert, update, and delete information in the database based on user input and perform any other SQL commands.

Active Movie provides next-generation, cross-platform video; the Active Movie Stream Format solves several important synchronization issues in multimedia-content storage and transmission.

Active Animation gives you a powerful foundation for Internet interactive, animated, and multimedia-based content, targeting domains such as advertising, entertainment, online shopping, and technical illustration.

Microsoft is building an infrastructure around the client/server model that enables secure transactions, billing, and user authentication. IE 2.0 and 3.0 supports the secure socket layer (SSL) 2.0 and 3.0 version and personal communications technology (PCT) 1.0 version. Most importantly, ActiveX is built on Win32 and OLE, which enables developers to build on their existing investments. ActiveX is the

doorway to a whole new world of Internet applications.

ActiveX Program Development

The most fundamental model that you should understand before embarking on ActiveX development is the COM (Component Object Model). This model is the same model as discussed in the OLE COM specification.

COM is the "Object Model" for ActiveX and OLE. Microsoft also provides OLE COM wizard and ActiveX Template Library (ATL) so that developers can develop lightweight, fast COM objects. ATL offers a template to write OLE automation server, OLE controls, and the very basic dual interface or any arbitrary COM objects. ATL also provides a custom AppWizard (called OLE COM AppWizard in VC 4.x project workspace) which can be used with Visual C++ 4.1 or later to create a COM object skeleton.

Microsoft Visual C++ 4.1 Development Studio integrates different AppWizards and Control Wizard to simplify the development process. Along with Visual C++ 4.x is version 4.x of the Microsoft Foundation Classes. 4.1. In Microsoft Visual C++ 4.2, there is new support for ActiveX programming, such as

- WinInet
- Active document
- Asynchronous moniker
- URL moniker
- Control for Internet

Besides the tools, wizards, frameworks, and foundation classes, Microsoft also provides a set of specifications to implement certain ActiveX controls. For instance, it provides the OLE controls for Internet, Document object, ActiveX scripting interface, Hyperlink interface, and Asynchronous Moniker specification.

ActiveX controls can be easily manipulated by any scripting languages in IE 3.0. The scripting languages include Java script and the client side VBScript and JavaScript, or any other third party scripting language that implements the ActiveX scripting interface. ActiveX controls, particularly OLE automation servers, can also be used with the server side VBScript.

Microsoft also provides J++ to develop the Java Applet and Java Applications. Java Applet can be used as an ActiveX controls.

Besides these ActiveX client side script, Microsoft also provides a system on top of the IIS 2.0 to use the server-side VBScript. ActiveX controls, particularly OLE automation server, can be referenced in the server-side VBScript.

A variety of tools can be used to develop server-side components, such as Perl and C for CGI programming.

To facilitate developing ISAPI applications, Microsoft Visual C++ 4.x provides the ISAPI Extension

Wizard and some foundation classes for ISAPI. Along with this, Microsoft also provides the ISAPI specification.

The Active Movie add-on tool kit includes tools to develop applications that handle streamed media. This tool kit allows software developers to integrate real-time audio and video content in virtually any type of application.

Active Animation and Active Movie controls can be manipulated through client-side VBScript. A developer can glue the control's functions together without the need for complex stream synchronization methods.

Summary

ActiveX is a technology that has the potential to change the way information is accessed and used on the Internet. Powerful abstractions based on OLE have been developed that enable fast, scaleable integration of your objects within the Internet. Microsoft is making a major effort to make the Internet everything it can possibly be. By using ActiveX, developers can make the best use of their system resources while providing instant, dynamic content and functionality in their Internet applications. How information is presented greatly affects how interesting and usable people find it.

Chapter 2

OLE Components

by Vincent W. Mayfield

Welcome to the exciting new world of ActiveX Programming! But is ActiveX really new? Well, yes and no. In Chapter 1, "An Overview of ActiveX," you learned the ActiveX themes. In addition, you also got an overview of what ActiveX can do for you. But what are the foundations of ActiveX? Moreover, what technologies does ActiveX include? These are the same questions I found myself asking when I returned from the Software Development 96 Conference last March.

During a lecture I attended, someone suggested that ActiveX was nothing more than Internet-aware OLE Controls. I started asking questions, and someone else informed me that ActiveX is nothing more than a sly marketing attempt to sell OLE under a different name. I was left perplexed and confused because no one could give me a definitive answer about the internals and framework of ActiveX or even an explicit definition of what ActiveX is. I decided to find out for myself just what ActiveX is.

What I found is that ActiveX is composed of a group of technologies or components to develop and implement applications for the Internet. I soon understood why no one could give me a clear definition. At the core of these technologies is OLE. ActiveX is an extension of OLE technologies across the Internet. But ActiveX is more than an extension of OLE—it also comprises a series of Internet and multimedia services that can be used to create rich Internet applications. However, to understand ActiveX, you must first understand OLE. So this is where this chapter's exploration of ActiveX begins.

To understand the ActiveX Control, you must first understand ActiveX and most importantly OLE. This chapter examines each component technology that makes up OLE. Next, it covers how each component that makes up OLE is extended across the Internet through ActiveX. Then you will investigate the new technologies in ActiveX. Lastly, you will examine the OLE/ActiveX technologies included in an ActiveX Control. Keep in mind during your reading that OLE, ActiveX, and Internet programming are not easily mastered. The key to the deployment of these technologies is a thorough understanding of their concepts.

OLE: An Introduction

In 1991, Microsoft introduced a new specification called OLE 1.0. The OLE in OLE 1.0 stood for object linking and embedding. OLE 1.0 was basically a way of doing compound documents. A *compound document* is a way of storing data in multiple formats, such as text, graphics, video, and sound, in a single document. *Object-oriented* was the new programming buzzword, and the OLE 1.0 specification was a move to a more object-oriented paradigm. Furthermore, OLE 1.0 was an effort to move toward a more document-centric approach, instead of an applications-centric approach. Unfortunately, OLE 1.0 was coldly received by software developers. Very few independent software vendors (ISVs) and corporations raced to embrace OLE 1.0 and OLE-enable their applications. This reluctance to deploy OLE 1.0 in applications was mainly because OLE 1.0 had a steep learning curve. In addition, OLE 1.0 had to be coded using a very complex C API, which embodied programming concepts new to most developers.

Fortunately, Microsoft continued to strive to improve OLE and in 1993, released the OLE 2.0 specification. This new specification encompassed more than just compound documents; it sported an entire architecture of object-based services that could be extended, customized, and enhanced. The foundation of this services architecture was the Component Object Model (COM). The services available through this architecture are

- COM
- Clipboard
- Drag and Drop
- Embedding
- In-Place Activation
- Linking
- Monikers (Persistent Naming)
- OLE Automation
- OLE Controls
- OLE Documents
- Structured Storage
- Uniform Data Transfer

From a programmatic view, OLE 2.0 is a series of services built on top of each other. These services form an architecture of interdependent building blocks built on the COM foundation.

Figure 2.1. OLE is built on the COM.

The release of OLE 2.0 had such an impact on standard ways of computing that it received two prestigious industry awards: a Technical Excellence award from *PC Magazine* and the MVP award for software innovation from *PC/Computing*. Adding to the OLE 2.0 success was a new and improved programming interface. Developers could now move to OLE-enabled applications much more easily. The OLE 2.0 services incorporate many of the principles embodied in object-oriented programming:

encapsulation, polymorphism, and an object-based architecture. Further adding to the success of OLE 2.0 was the release in February of 1993 of Visual C++ 1.0 with the Microsoft Foundation Class (MFC) Library version 2.0. MFC had wrapped the OLE API in a C++ class library, thus making it much easier for programmers to utilize the OLE services architecture.

Note

Don't let the ease of use of the MFC Library fool you. OLE programming is very difficult to master. However, I recommend that fledgling OLE and ActiveX programmers utilize MFC. MFC provides a framework to get you up and programming very quickly. Trying to program at the API level initially can lead to frustration and discouragement. If you do not know OLE, my advice is to learn the services, concepts, and standards and then go back and understand the low-level C API. Understanding the services, their interfaces, and when to use them is the main key to OLE and ActiveX programming.

Today OLE is no longer an acronym. The term object linking and embedding is now obsolete. Microsoft refers to it as simply OLE. Notice that there is no version number attached to OLE any more. Because OLE is an extensible architecture, it can be enhanced and extended without changing its basic foundation. A testimonial to this capability is OLE Controls. OLE Controls were not part of the original release of OLE. OLE Controls were not added to the available OLE services until almost a year after the original release. In fact, objects created with OLE 1.0 still work and interact with modern OLE applications. However, their functionality is limited to the original 1.0 specification. Thus there is no need for versions. From here on out, this chapter will simply refer to OLE unless specifically outlining a feature of OLE 1.0.

OLE: The User's Perspective

Because the end user is the main reason software is developed, this section will view OLE from the user's eyes. This will help you to grasp the benefits and the available services of OLE and ActiveX. The end user's view is simple, less technical, and very understandable. I firmly believe that the user decides in the first ten minutes of using an application whether he or she likes it. This sets the stage for all further experiences utilizing that application. Therefore, an application's intuitiveness, appearance, ease of use, capability of performing work or entertaining, and performance are of paramount importance.

Always keep in mind that the "devil-spawned end user", as the cartoon character Dilbert, by Scott Adams would say, is the main reason for our existence as software engineers. The best software engineers never forget this and always tackle every programming endeavor with the end user in mind.

Microsoft—and Apple before them—knew this. This is why Windows and the Macintosh each have a standard interface, not only from a user's perspective, but also from a programmer's perspective. Users interact with OLE in three ways: OLE Documents, OLE Automation, and OLE Controls. As a computer professional, you have seen or worked with Microsoft Word or Excel before. Microsoft Word is the

classic example of an OLE Document. This chapter is not going to outline the functionality of Word but merely point out the features of OLE. However, do not be deceived; OLE Documents are not always the classic word processor. It is easy to think so because of the word "documents."

The first feature of OLE Documents is a common user model. This simply means that the User Interface (UI) features used to access OLE Documents are similar from application to application. The common user model features document-centricity and takes advantage of OLE's integrated data capabilities.

Note

OLE user-interface-design guidelines are well documented in *The Windows Interface Guidelines for Software Design*, Microsoft Press, 1995.

One of these integrated data capabilities is called Linking and Embedding. Data objects of different types, created from other applications, can be embedded or linked into an application's OLE Document. This enables the user to manipulate the object in the host application without returning to the creating application. The object is simply edited in place, hence the term "in-place editing." The user interface is modified in the host application with menus, toolbars, and context menus from the application that created the object.

In Figure 2.2, take note of the two kinds of data, text and an embedded Visio drawing. Also note the toolbars and menu.

Figure 2.2. Microsoft Word document with embedded text and graphics.

If you double-click the Visio Drawing, the Word application changes, and new user interface objects are added to Word from Visio (see Figure 2.3). Notice that the Word user interface performs a metamorphosis and now has the Visio tool bars, floating dialog boxes, and menu items, as well as the Visio drawing and rulers. The user can then edit this drawing object without switching applications. In addition, these objects can be dragged and dropped between and within applications.

Figure 2.3. Microsoft Word document with the Visio drawing activated for in-place editing.

These features are implemented in the same way from application to application. Thus, there is a smaller learning curve for users when they get new applications, because the applications function similarly and have a common user model.

The next level of visibility to the user is OLE Automation. OLE Automation enables the user to access and modify objects through properties and methods utilizing a high-level language like Visual Basic for Applications (VBA). This enables the user to customize objects, and the inter-activity between objects, to perform operations the way the user defines. Microsoft Excel spreadsheets are the classic OLE Automation objects. The user can create Excel spreadsheets that update a Microsoft Graph object or update information in a Microsoft Access or Borland Paradox database. The greatest part of OLE Automation is that you do not have to be a programmer to take advantage of it. This is done through VBA. Microsoft has made VBA easy to learn using a Macro Recorder (see Figure 2.4) that records your keystrokes in VB code and an "object browser" that you use to paste the proper code where you need it. Anyone can learn to use it.

Figure 2.4. Microsoft Excel with the Macro Recorder invoked.

OLE Controls are the last area of OLE visibility to the end user. They are self-contained, reusable components that can be embedded in applications. To the user, they are nothing more than a control that takes their input and passes it to the application that contains it. However, some OLE Controls are static in nature, such as a picture control. OLE Controls are also OLE Automation objects that can have properties set at both compile time and runtime, and OLE Controls also have methods that can perform certain operations. The difference between OLE Controls and OLE Automation objects is that they are self-contained objects. They provide two-way communication between the control and the container. In future sections, you will discover that OLE Controls have been extended to ActiveX Controls that can be utilized across the Internet. These components have a very profound impact in the area of application development (see Figure 2.5), because they are prebuilt. From the end user's perspective, they provide increased functionality and lower software costs.

Figure 2.5. Properties and methods of an OLE Control during development in the Visual C++ Microsoft Developer studio.

OLE Services: A Programmer's View

This section covers the OLE Services from a programmatic view. For each service, you will be given a description of the technology and a programmer's view of the interfaces to these OLE services. Pay particular attention to understanding what each service does and where it fits into the architecture. The explanations highlight the interfaces to these objects and some key properties and methods where appropriate. Some of these services will be discussed in detail in later chapters as they pertain and integrate into ActiveX. Remember that OLE consists of the following services:

- COM
- Clipboard
- Drag and Drop
- Embedding
- In-Place Activation
- Linking
- Monikers (Persistent Naming)
- OLE Automation
- OLE Controls
- OLE Documents
- Structured Storage
- Uniform Data Transfer

Notice these are the same technologies the end user sees. However, the end user's view is a visual one, and the programmers view is a menagerie of interfaces that must be mastered to provide the slick visual

representation the end user sees. As discussed earlier, these services form building blocks on which each element in the architecture builds upon, as shown in Figure 2.6.

Figure 2.6. A programmatic view of OLE.

This architecture starts with the foundation, the COM.

Component Object Model (COM)

When Microsoft designed OLE, it was designed with object-oriented programming in mind. COM objects are much like instantiated C++ classes or an ADA Package. In fact, COM was designed with C++ programmers in mind. It supports encapsulation, polymorphism, and reusability. However, COM was also designed to be compatible at the binary level and therefore has differences from a C++ object. As a programmer, you are aware that compiled programming languages such as C, C++, PASCAL, and ADA are machine-dependent. As a binary object, a COM object concerns itself with how it interfaces with other objects. When not utilized in the environment of its creator, an interface is exposed that can be seen in the non native environment. It can be seen because it is a binary object and therefore not machine-dependent. This does not require the host environment or an interacting object to know anything about the COM object. When the object is created in the womb of its mother application, COM does not concern itself with how that object interacts within it. This interaction is between the mother application and the child object. It is when the object interacts with the rest of the world that COM is concerned about how that object can be interfaced with. It is important to note that COM is not a programming language: it is a binary standard that enables software components to interact with each other as objects. It is also a programming model to facilitate the programmability of this standard.

COM objects consist of two types of items, *properties* and *methods*. Properties are the data members, and methods are member functions. COM objects each have a common interface. No matter what they do, COM objects all have to implement the IUnknown interface. This interface is the main interface for all others. The IUnknown interface has the following member functions:

- ULONG AddRef(void)
- ULONG Release(void)
- HRESULT QueryInterface(REFIID id, void **ipv)

Each object implements a *vtable*. A vtable is nothing more than an array of pointers to member functions implemented in the object (see Figure 2.7). This vtable is shared between all the instances of the object also maintaining the private data of each object. A client application evokes an instance of the interface and gets a pointer to a pointer that points to the vtable. Each time a new interface to the object is instantiated, the reference count of objects is incremented with AddRef(). Conversely, each time a reference is destroyed, the reference counter is decremented with Release(). Once the reference count is zero, the object can be destroyed. In order to see what interfaces an object supports, you can use QueryInterface().

Figure 2.7. How an interface maps into a vtable.

This section has covered a general overview of COM. An in-depth review of COM and how COM is implemented through ActiveX, complete with examples, is presented in Chapter 3, "Creating COM Objects."

Structured Storage

Unfortunately, most platforms today have different file systems, making sharing data a very difficult task. In addition, these file systems arose during the mainframe days when only a single application needed to access the disk at any one time. COM is built with interoperability and integration between applications on dissimilar platforms in mind. In order to accomplish this, COM needs to have multiple applications write data to the same file on the underlying file system. OLE Structured Storage addresses this need.

Structured Storage is a file system within a file. Think of it as a hierarchical tree of storages and streams. Within this tree, each node on the tree will have one and only one parent, but each node may have from zero to many children. Another way to think of it is like the Windows 95 Explorer. The folders are the storage nodes, and the files are the streams. Structured Storage provides an organization chart of data within a file as seen in Figure 2.8. In addition, this organization of data is not limited to files, but includes memory and databases.

Figure 2.8. Structured Storage is a hierarchical tree of storages and streams.

Stream objects contain data. This data can be either native data or data from other outside objects. Storage objects are compatible at the binary level; thus, in theory, they are compatible across platforms. However, you all know that there are minute differences between the various platforms. Notice in Figure 2.8 the tree of the structured storage object. The definition of the tree is dependent on how the objects creator defined the storage of the object.

Structured Storage objects are manipulated utilizing the following OLE Interfaces:

- IPersistStorage
- IStorage
- IStream

IStorage,as the name implies, manipulates storage objects. Likewise, IStream manipulates streams. Rarely would you want to manipulate stream or storage objects individually. More than likely, you would want to manipulate the persistent storage object with the IPersistStorage. By persistent storage, I mean data that will continue to exist even after an object is destroyed. For example, if you wanted and allowed the user to define the color of an object such as a text label, you would persistently store that object's foreground and background colors. The next time the object was created you could read in from persistent storage the colors previously chosen by the end user. You could then apply those attributes to the object and thus maintaining the users preferences. IPersistStorage enables you to do this by performing the following operations:

- IsDirty

- InitNew
- Load
- Save
- SaveCompleted
- HandsOffStorage

A great way to see what structured storage looks like is with a utility that comes with Visual C++ 4.X called DfView. DfView is in the \MSDEV\BIN directory of Visual C++. DfView enables you to look at a compound file also known as an OLE Document. OLE Documents implement structured storage. Figure 2.9 shows an example of DfView (this is the Word document with an embedded Visio drawing object seen earlier in Figure 2.2).

Figure 2.9. DfView shows the hierarchical tree of a structured storage object.

If you double-click a stream object, you can see its binary contents (see Figure 2.10).

Figure 2.10. The binary contents of a stream object.

Monikers (Persistent Naming)

Monikers are a way to reference a piece of data or object in an object-based system like OLE. When an object is linked, a moniker is stored that knows how to get to that native data. For example, if you link a sound file into a Word document, the .WAV file is not stored natively in that document. A moniker is created that can intelligently find the .WAV file object.

To utilize a moniker to locate and bind to an object, you must utilize the IMoniker interface and call IMoniker::BindToObject. By utilizing the intelligent persistent name of that object, the IMoniker interface negotiates the location of that object and returns a pointer to the interface of that object's type. The moniker itself then dies. Think of it as similar to de-referencing a pointer in C or C++ to locate a piece of data. Remember that monikers are persistent. IMoniker is derived from IPersistStream, and thus it can serialize itself into a stream, hence persistence. There are five basic types of monikers:

- File monikers
- Item monikers
- Anti monikers
- Pointer monikers
- Composite monikers

File Monikers

File monikers store a filename persistently. In binding the text filename to the file object, a pointer to the file object interface is returned so that you can manipulate that file object.

Item Monikers

Item monikers point to a specific place inside a file, such as a paragraph or a portion of an embedded video.

Anti Monikers

Anti monikers delete the last moniker in a series or chain of monikers, as in a composite moniker.

Pointer Monikers

Pointer monikers simply point to other monikers wrapping them in a chain. However, it should be noted that pointer monikers are not persistent.

Composite Monikers

A composite moniker is an ordered collection of monikers. At the root of a composite moniker is a file moniker that references the document path name. It then holds a series of item monikers. Composite monikers are used when you need to have a collection of monikers within a single object.

Uniform Data Transfer (UDT)

Through OLE, you can utilize structured storage to store your objects, and monikers to find your objects, but there has to be a mechanism to move this data from the place it is stored (linked or embedded) to where you can output it to the client for manipulation. In addition, Uniform Data Transfer (UDT) also notifies the data object and the client of changes in the data. Uniform data transfer provides this service through the IDataObject interface. UDT is used primarily in three areas:

- ClipBoard
- OLE Drag and Drop

- Linking and Embedding

Clipboard

The system clipboard is a system-level service used for interprocess communications. Because it is a system-level service, all applications have access to it. OLE can utilize the clipboard to do UDT of objects between processes. With an IDataObject pointer, you can use the function OleSetClipboard() to take a cut or copied object and expose this object to all processes through the clipboard. Likewise, when you want to paste data from the clipboard, you can use your IDataObject pointer to utilize the OleGetClipboard() function. This is a very powerful mechanism because it maintains the integrity of the object as a whole, enabling you to move complex object data types between applications.

Drag and Drop

Drag and Drop is a method by which the user can select and move objects within an application and between applications. UDT is used to perform Drag and Drop actions. On the selection of the object, the source application packages the object and uses an IDataObject pointer to call DoDragDrop(). The source uses the IDropSource interface, which yields a pointer to its implementation. This pointer is passed to DoDragDrop(). The source controls the mouse cursors and handles the object in case of a cancellation.

Once the user brings the dragged object to its new client location or target, the client application evokes the IDropTarget interface. With the pointer to the IDropTarget, the client application tracks the object in relation to itself with the functions available in the IDropTarget interface. One function called IDropTarget::Drop() is called when the object is dropped on the target. Drop() passes the IDataObject pointer of the source to the target. Now that the client has the IDataObject pointer, it is free to manipulate the object.

Embedding and Linking

A *linked* object is an object that is not stored within an OLE Document. In the document, a moniker is stored that references the linked object. This OLE function utilizes UDT to move the data from the data object source to the container application so that the data can be rendered as appropriate. Linked objects are manipulated through the IOleLink interface. By linking an object instead of embedding it, you cut down on the size of the compound file. In addition, you expose the linked object so that multiple people can utilize it.

An *embedded* object is an object that is stored, through the OLE structured storage mechanism, as native

data within an OLE Document. Although this increases the size of the compound file, it provides a single file object that can contain multiple data types.

OLE Documents

OLE Documents are nothing more than compound files that utilize structured storage to hold the objects that make up the document. These objects that make up the document can be native data, or they can, through the use of monikers, link to data outside of the document. In addition, an OLE Document can contain objects created by other processes, embedded as if they were natively a part of the document. OLE Documents are handled through interfaces just like any other OLE object. As you can see, OLE Documents are a conglomeration of several OLE services. Here are some of the interfaces utilized to implement OLE Document interfaces:

- IOleItemContainer
- IPersistFile
- IClassFactory
- IOleInPlaceActiveFrame
- IOleInPlaceUIObject
- IOleInPlaceSite

In-Place Activation

OLE Documents support in-place activation or what is commonly referred to as visual editing. This enables you to edit embedded objects in a container application as if they were native. When you activate visual editing in the container, the user interface of the container morphs to support selected user-interface functions of the server application that created the object. There are a whole series of interfaces to enable you to implement and support in-place activation. These interfaces all begin with IOleInPlace. These are some of the interfaces you can utilize to implement and support in-place activation:

- IOleInPlaceObject
- IOleInPlaceActiveObject
- IOleInPlaceSite
- IOleInPlaceActiveFrame
- IOleInPlaceUIObject
- IOleInPlaceSite

OLE Automation

OLE Automation basically enables you to manipulate the properties and methods of an application from within another application through the use of high-level macro languages and scripting languages like VBScript and JavaScript. This enables you to customize objects and provide interoperability between applications.

In the world of OLE Automation, there are OLE Automation Components and OLE Automation Controllers. An OLE Automation Component is a service that is exposed by an application for use by another. Microsoft Excel is a good example of this, as it exposes services that can create and manipulate worksheets, cells, and rows.

What services are available through an OLE Automation Component are stored in a type library. A type library is stored in a binary file with a TLB extension. Object Description Language is used to define the services of an OLE Automation Component. Object Description Language instructions are stored in a file with the extension ODL. The ODL file is compiled into a type library. In Visual C++ and in the ActiveX SDK, there is a nice utility that reads type libraries and graphically displays the services provided by OLE Automation Components.

The utility in Visual C++ is called OLE 2 View 32 application. The utility in the ActiveX SDK is called OLE\COM Viewer, and it is a newer implementation than the OLE 2 View 32 application in Visual C++. Figure 2.11 shows the OLE/COM Viewer, which can be used to view OLE and COM objects graphically.

Note

The OLE2View32 Application in Visual C++ is located in the \MSDEV\BIN\ directory, and the filename is OLE2VW32.EXE. The OLE\COM Viewer in the ActiveX SDK is in the \INETSDK\BIN\ directory and the filename is OLEVIEW.EXE.

Figure 2.11. The OLE/COM Viewer that comes with the ActiveX SDK.

Notice that the Type Library Viewer screen (see Figure 2.12) shows the disassembled type library in Object Description Language. It also displays the constants, properties, methods, and interfaces to the Automation Component.

Figure 2.12. The OLE/COM Viewer's function Type Library Viewer.

OLE Automation Controllers are applications that use the services provided by OLE Automation Controllers. OLE Automation Controllers work through an interface called IDispatch. This dispatch interface exposes the available services to the controller application.

OLE Controls

As discussed previously, OLE Controls are self-contained reusable components that can be embedded in applications. OLE Controls are also OLE Automation objects that can have properties set at both compile time and runtime, and OLE Controls also have methods that can perform certain operations. The difference between OLE Controls and OLE Automation objects is that they are self-contained objects. They provide two-way communication between the control and the container. These components have a very profound impact in the area of application development. These reusable self-contained pockets of functionality are discussed in detail in Chapter 5, "Creating OLE Controls."

OLE Technologies Extended through ActiveX

ActiveX has taken the OLE Technologies and extended them beyond the bounds of the local machine, to Enterprise Wide networks and the Internet. Specifically, OLE Technologies have aggrandized into the following ActiveX services:

- ActiveX Documents
- ActiveX Controls
- COM
- Internet Monikers

This is not the total affect. Elements of OLE are also present in the new ActiveX technologies, as will be discussed in the next section, "ActiveX Technologies." For now we will concentrate our discussion on the evolution of OLE Technologies into ActiveX.

ActiveX Documents

ActiveX has taken OLE Documents and extended them across the Internet. This technology is a way for existing OLE Documents such as Microsoft Word, Microsoft Project, and Microsoft PowerPoint to be activated by a Web browser and brought up through a viewer. Thus you can have compound files with various data that can contain linked and embedded objects being accessed across the World Wide Web (WWW). Utilizing the ActiveX Hyperlinks technology, you can extend OLE Documents across the Web. ActiveX Hyperlinks are discussed in the next section. In addition, ActiveX Documents are discussed in depth in Chapter 19, "OLE Document Objects."

Asynchronous Storage

The ability to bring ActiveX Documents across the WWW gives rise to another ActiveX technology, Asynchronous Storage. Basically this extends structured storage across the Web, allowing for the storage to happen asynchronously. Obviously, with the slow bandwidth of the Internet, if you allowed a storage

operation to happen synchronously, nothing else could happen on the client or server until the transfer of data to or from persistent storage took place. Utilizing ActiveX Hyperlinks and the technology of Asynchronous Monikers, Asynchronous Storage is accomplished.

ActiveX Controls

ActiveX Controls are simply OLE Controls or OCXs that have been extended to the Internet environment. Microsoft has now replaced the term *OLE Control* with ActiveX Control. Remember, OLE is an extendible architecture; therefore, these reusable components can not only be embedded in a Web page, but also in a non-Internet-enabled application. ActiveX Controls are covered in depth in Chapter 5, "Creating OLE Controls."

ActiveX Controls can be created in a variety of languages, including C, C++, Java, and according to Microsoft, the next release of Visual Basic. They can also be manipulated though VBScript or JavaScript, so you do not even have to be a programmer to use them.

ActiveX Controls are great components as you have a virtual plethora of little pockets of prefabricated functionality you can take advantage of. The possibilities for ActiveX Controls are endless. Currently, ActiveX Controls range from a Calendar Control to a Picture Control that enables you to display static pictures.

COM is at the base of the ActiveX Control technology. ActiveX Controls are built on a series of OLE services, with COM as the base. The following list depicts the technologies that are encompassed in the ActiveX Control:

- Component Object Model
- Connectable Objects
- Uniform Data Transfer
- OLE Documents
- Property Pages
- Persistent Storage
- OLE Automation

ActiveX Control: COM

Just like the OLE Controls previously discussed, ActiveX Controls are COM objects. They are in-process OLE Automation servers activated from the inside out. Like every other COM object, they expose the IUnknown so that container applications can access their properties and methods through the pointers returned by the interface.

ActiveX Control: Connectable Objects

ActiveX Controls support two-way communication from the control to the client application. This method of communication is called Connectable Objects. It enables the control to notify the client of events or invoke a method or event. It also enables the client to communicate directly with the control.

ActiveX Control: Uniform Data Transfer

Controls can be dragged and dropped within their client application if that functionality is enabled in the client application.

ActiveX Control: Compound Documents

In the beginning of this chapter, you saw how an object from another application could be embedded in a host application. In addition, that object could be in-place activated for visual editing. Likewise, OLE Controls are built on the concept of OLE Documents and can be in-place activated.

ActiveX Control: Property Pages

ActiveX Controls have property pages, like their predecessor OLE Controls, that expose its properties and methods to the user. From the property pages, the properties can be set.

ActiveX Control: OLE Automation

ActiveX Controls are automation servers. Their properties and methods can be set at compile time through the use of property pages and at runtime through VBScript and JavaScript.

ActiveX Control: Persistent Storage

COM objects can utilize Persistent Storage in a variety of ways. ActiveX Controls utilize Persistent Storage to store their state. This enables the control to be initialized to the state it was when you last

utilized it.

COM

As you learned previously, COM is a binary standard for objects. Basically COM operates the way it did before ActiveX, except that COM has been extended so that you can exchange and utilize objects across the Internet. This has given rise to Distributed COM.

Distributed COM (DCOM)

Distributed COM, also known as DCOM and formerly known as Network OLE, is the basic extension of binary COM objects across LANs, WANs, and the Internet. Now you can instantiate and bind objects across a network. A detailed analysis of COM is presented in Chapter 3, "Creating COM Objects."

Internet Monikers

With the advent of ActiveX and the extension of COM across the net, quite naturally monikers were also extended and incorporated into this architecture. This gave rise to two new types of monikers.

- URL monikers
- Asynchronous monikers

URL Monikers

A URL is a universal resource locator, used for Web-based addressing of objects. As you learned earlier, monikers are an intelligent naming system, so that by utilizing the IMoniker interface to an moniker object and the intelligent name, you can locate the object. This capability was simply extended to include URLs because of the capability to pass objects across the net from DCOM.

Asynchronous Monikers

Previously, monikers carried out their binding to the object synchronously. Nothing could happen until the binding was complete. On the high latency, slow-link communications network of the Internet,

holding up operations while binding is accomplished is unacceptable. Thus with asynchronous monikers, the interfaces to the object negotiate the transmission of the binding process, so as to perform it asynchronously. Right now, URL monikers are the only implementation of asynchronous monikers.

ActiveX Technologies

ActiveX brings to the table some new technologies that are not necessarily related to OLE. However, these technologies facilitate the creation of interactive applications for the World Wide Web. These items are

- ActiveX hyperlinks
- ActiveX conferencing
- ActiveX server extensions
- Code signing
- HTML extensions
- ActiveMovie

ActiveX Hyperlinks

ActiveX hyperlinks basically allow in-place activation from HTML files of non-HTML based documents. Utilizing an ActiveX document container, you can access Microsoft Word, Microsoft Excel, Microsoft PowerPoint, Visio, and CorelDraw! documents from a hypertext link in an HTML document.

ActiveX Conferencing

The ActiveX conferencing services are a suite of technologies that enable real-time, multiparty, multimedia communication over the Internet. This is much like video teleconferencing except you can do it on a PC. Just think what this does for programmers; we could all work at home and telecommute. This is a programmable interface opening up endless possibilities for innovation.

ActiveX Server Extensions

ActiveX server extensions, formerly known as the ISAPI Internet Server API, are used to give functionality to Internet servers. Previously this could only be done utilizing common gateway interface

(GCI) code. ActiveX server extensions provide an alternative means to achieve this functionality. Usually server extensions are implemented utilizing a dynamic link library (DLL) and provide some functionality not provided by the HTTP server, such as connecting to a database.

ActiveX Scripts

ActiveX scripts bring OLE Automation to the Internet. Automation Controllers can now access Automation Component Services across the Internet with DCOM and ActiveX support for scripting. You can use a variety of scripting languages such as VBScript, JavaScript, Perl, Visual Basic for Applications, Lisp, and Scheme. To explore scripting, see Chapters 8, "Microsoft Internet Explorer Object Model for Scripting," 9, "VBScript," and 10, "JavaScript."

Code Signing

Code signing is a new technology that enables electronic signatures for code. This provides security from tampering of interactive applications across the net. Basically, the application vendors will provide a digital signature for their code that compiles with the Code Signing specification. On the client side, when an application or component is downloaded from the net, it calls a Win32 API function called WinVerifyTrust(). This function checks the digital signature and verifies it.

HTML Extensions

HyperText mark up language (HTML) is the language for all Web-based document production. In order to support ActiveX Controls and ActiveX Scripts, extensions had to be made to the HTML language. In addition, Web browsers had to be modified to accommodate the new language extensions. Now you can add ActiveX Controls to Web pages using the HTML <OBJECT> tag. These new extensions will be covered in Appendix O, "HTML Extensions."

ActiveMovie

ActiveMovie is a new technology to replace the old Media Control Interface and Video for Windows. ActiveMovie is an audio and video steaming framework. With ActiveMovie you will be able to play back MPEG, AVI, and Apple Quicktime movies.

Summary

This chapter has discussed OLE and the component architecture that makes up OLE. Because OLE is an object-oriented architecture founded on the COM, it is an extendible services architecture. Each OLE component is a building block for the rest of the technologies. Microsoft has extended this architecture to ActiveX to facilitate the creation of Internet-enabled applications. ActiveX builds on OLE and COM and adds new technologies of its own. ActiveX Controls are Internet-aware controls that are nothing more than an extension of the OLE Control architecture and make it easy to extend the use of reusable components on the Internet.

Next, Chapter 3 explores the COM in depth and shows you how to create COM objects. You will find out why COM objects, as opposed to C++, ease the programming challenge.

Chapter 3

Creating COM Objects

by Weiying Chen

The Component Object Model(COM) is an open architecture for cross-platform development of client/server applications. It is the cornerstone for ActiveX technology and OLE 2.0.

This chapter presents the fundamental concept of COM, such as COM client/server architecture, COM server, and COM client. A set of fundamental COM interfaces are also examined in detail to describe their roles in creating COM objects.

To illustrate the fundamental blocks and the concept of the COM architecture, serial examples are built step-by-step to demonstrate how to create various COM servers and corresponding COM client applications. All source codes in examples are written in Microsoft VC++ 4.1.

Microsoft Active Template Library (ATL) simplifies the procedure of creating COM servers by providing commonly used templates. At the end of this chapter, an example demonstrates ATL creating COM objects.

COM Client/Server Architecture

The Component Object Model (COM) is an open architecture for cross-platform development of client/server applications. It is the cornerstone for ActiveX technology and OLE 2.0 as shown in Figure 3.1.

Figure 3.1. COM, ActiveX and OLE 2.0.

All ActiveX controls are COM objects. COM objects refer to any object that implements IUnknown interface.

ActiveX scripting provides a set of OLE interfaces for a scripting engine and a scripting engine host. All these interfaces inherit from IUnknown.

ActiveX Document provides a set of Document object and Document object container interfaces, which inherit from IUnknown interfaces.

ActiveX server-side scripting uses OLE technology.

OLE 2.0 is built on COM.

COM provides a client/server architecture. The COM client uses COM server via the COM library. Figure 3.2 illustrates.

Figure 3.2. COM client/server model.

COM Server

A COM server is a component that implements one or more COM Class objects. A COM Class object is a COM object that is creatable via a COM Class Factory object. A COM Class object has a CLSID associated with it. A COM object is anything that implements IUnknown interface. It is different from an object in object-oriented programming. In Object-Oriented programming, an object called OO object here is an entity that has state, behavior and identity. The OO object's state is represented by the value of the attributes in the object. But a COM object's state is implied by the interface; the state is not explicitly stated, because there are no public attributes exposed in the interface. The Interface is just a set of functions without any attributes.

The OO object's behavior is a sequence of messages sent to the object, which is a sequence of methods called on this object. But for a COM object, the object's behavior is defined as the interface it supports.

The OO object's identity is a way to look at the object, whereas for a COM object, the identity is defined by moving between interfaces exposed by the COM object, this is done by invoking IUnknown::QueryInterface interface.

Each COM object provides functionality via exposing interfaces. An interface is a group of related functions and provides some specific service. For example, the COM server in Figure 3.3 exposes two interfaces, one is IPrint, the other is IHelp. IPrint interface provides the print service, whereas IHelp supports the help service. Each interface groups its own functionality.

Figure 3.3. CPrint COM server.

In order to uniquely identify the class object provided by the COM server, a class identifier(CLSID) is used, whereas to identify the interface, an interface identifier(IID) is used.

A COM server is usually a .DLL or .EXE file. A DLL based COM server is called an in-process(in-proc for short) server because it loads into the same address space as the client. The client can make direct call to the object, which is faster and more efficient. But the crash of the DLL can destroy the client's address space.

An EXE based COM server is called an out-process(out-proc for short), because it runs in its own separate process space.

An EXE based COM server isolates from the address space of the caller, which makes it more reliable. If the server crashes, it will not destroy the address space of the client. But because it is in a separate process, all interface calls must be marshaled across process(data gets copied), which affects the performance.

Note: Now with Java, COM server could be a Java class.

COM Client

A COM client(client for short) is an application that uses COM server. A COM client asks COM to instantiate object in exactly the same manner regardless of the COM server types. This is done by invoking the COM function CoCreateInstance. After COM client retrieves the first pointer to the COM object, it can not distinguish from the interface whether the COM server being used is an in-proc, or out-proc server.

COM Client is an executable(EXE) application as compared with COM server that can be DLL based.

COM Library

The COM library provides an implementation of the Application Programming Interface (API). The specification also defines a set of interfaces that will be used by different COM objects.

The component in COM also supports the communication establishment between the client and server. This component provides location transparency for the client. In other words, the client does not need to know where the server locates; all these are taken care of by the COM library.

COM Library APIs Functions

COM library API functions provide the functionality to the COM applications. COM applications are any application that uses COM. The following gives an example of API functions.

- CoInitialize: Initialize the COM library. The COM library must be initialized before calling its functions. This API should be called when the COM application starts.
- CoUninitialize: Uninitialize the COM library. This will free all the maintained COM resources and close all RPC connections. This API should be called when the COM application exits.
- CoCreateInstance*:* Create an instance of the object class. This API is a helper function. A helper function means a function that encapsulates other functions and interface methods defined in the COM specification. CoCreateInstance wraps the following sequence calls: COM API CoGetClassObject, IClassFactory method CreateInstance, and IClassFactory method Release.

COM Fundamental Interfaces

COM predefines a set of interfaces to be used by client/server applications. Among these, IUnknown and IClassFactory interfaces are the most fundamental ones. IUnknown interface is required for any COM object. The QueryInterface method in IUnknown interface allows the client to access the object's identity and move between interfaces.

A class factory object is required for every object identified by a given CLSID. A class factory object implements the IClassFactory interface.

IUnknown Interface

IUnknown is the interface that any other interfaces inherit from. In other words, every interface except IUnknown inherits from IUnknown. Listing 3.1 illustrates the IUnknown interface definition.

Listing 3.1. IUnknown interface.

```
interface IUnknown
```

```
{
    HRESULT QueryInterface([in] REFIID riid, [out] void **ppv);
    ULONG AddRef();
    ULONG Release();
}
```

COM object must implement this interface. COM client will invoke the methods in the interface implemented by the COM object.

QueryInterface Method

QueryInterface provides the mechanism by which a client, having obtained one interface pointer on a particular object, can request additional pointers to other interfaces on the same object. The COM object exposes itself via a set of interfaces.

There are two parameters for QueryInterface. riid is IID of the interface requested. ppv is a return value. It is an indirect pointer to the interface. If the interface requested does not exist, ppv must be set to be NULL and an E_NOINTERFACE error code should be this method's return value.

The Listing 3.2 demonstrates an implementation of the QueryInterface method for CLowerStr class.

NOTE

The code listings in this chapter are from the example created in this chapter.

Listing 3.2. Example of QueryInterface Implementation

```
STDMETHODIMP CLowerStr::QueryInterface(REFIID iid, void **ppv)
{
    HRESULT hr;
    *ppv = NULL;
    if((iid == IID_IUnknown) || (iid == IID_ILowerStr) )
    {
        *ppv = (ILowerStr *)this;
        //increase reference count
        AddRef();
        hr = S_OK;
    }
    else
    {
        //if interface does not exist, *ppv set to be NULL, and E_NOINTERFACE
returns.
```

```
        *ppv = NULL;
        hr = E_NOINTERFACE;
    }
    return hr;
}
```

Here, CLowerStr is an implementation of the ILowerStr interface, ILowerStr interface inherits from IUnknown. CLowerStr can be called a COM object because it implements IUnknown interface.

AddRef Method

AddRef method provides the technique for an object to keep track of the reference count. The reference count should be incremented whenever an interface pointer is queried.

Listing 3.3 shows an implementation of the AddRef method.

Listing 3.3. AddRef method implementation.

```
STDMETHODIMP_(ULONG) CLowerStr::AddRef()
{
    m_dwRef++;
    return m_dwRef;
}
```

m_dwRef is a reference count defined in the object CLowerStr. It is defined as a DWORD.

Release Method

The Release method decrements the reference count. If the reference count is zero, the object should destroyed since the object is no longer needed. The client application needs to invoke this method whenever the interface is not accessed.

Listing 3.4 demonstrates an implementation of the Release method.

Listing 3.4. Release method implementation.

```
STDMETHODIMP_(ULONG) CLowerStr::Release()
{
    m_dwRef--;
    if(m_dwRef == 0)
        delete this;
    return m_dwRef;
```

```
}
```

IClassFactory Interface

IClassFactory is the interface that class factory object inherits from. In other words, class factory implements the IClassFactory interface. Class factory object is required in COM to create an instance of the object. It is a rule. For example, when the client application uses the CLowerStr object, CLowerStr object has to be created via its class factory.

Look at Figure 3.4; COM server has a class factory, which creates an instance of the object.

Figure 3.4. Relationship between Class Factory object and object.

IClassFactory interface has two fundamental methods shown in Listing 3.5.

Listing 3.5. IClassFactory Interface

```
interface IClassFactory : IUnknown

{

    STDMETHODIMP CreateInstance(IUnknown *punkOuter, REFIID riid, void **ppv);

    STDMETHODIMP LockServer(BOOL fLock);

}
```

The CreateInstance method creates an instance of the object class. It has to be implemented by the class factory object to instantiate the object. This method will be used inside the CoCreateInstance function call. CoCreateInstance will first return a pointer to the IClassFactory and then invoke IClassFactory's CreateInstance method to create an object's instance and returns an indirect pointer to the object's requested interface. This method only needs to be implemented but never needs to be invoked by the application itself.

punkOuter indicates whether the object is being created as part of the aggregate. If there is no aggregation in the COM server, NULL should be provided, otherwise, a pointer to the controlling IUnknown of the aggregate should be provided.

riid is the IID of the interface queried by the client. If the punkOuter is NULL, the IID of the initializing interface should be provided. Otherwise, riid must be IUnknown.

ppv is a pointer to the pointer of the requested interface. If the object does not support the interface specified in riid, ppv should be set as NULL, and E_NOINTERFACE should be returned as the method's return value.

LockServer locks the server in memory. The class factory will be revoked when the lock count is decremented to zero. LockServer(TRUE) will increment the lock count and ensure that the class factory will not be revoked.

Listing 3.6 illustrates an example of the implementation of CreateInstance.

Listing 3.6. Sample CreateInstance method implementation.

```
STDMETHODIMP CLowerStrClassFactory::CreateInstance (IUnknown *pUnkOuter,REFIID
iid,void **ppv)

{

   HRESULT hr;

   CLowerStr *pObj;

   *ppv = NULL;
```

```
   pObj = new CLowerStr;
   if (pObj)
   {
        hr=pObj->QueryInterface(iid,ppv);
        pObj->Release();
   }
   else
   {
       hr = E_OUTOFMEMORY;
        *ppv = NULL;
   }
   return hr;
}
```

CreateInstance first instantiates the CLowerStr object and then queries whether the iid interface exists in the CLowerStr object. If yes, ppv will return an indirect pointer to the interface, and the CLowerStr object will be released.

Listing 3.7 illustrates how to implement LockServer method.

Listing 3.7. Sample LockServer method implementation.

```
long g_cLock = -1;
STDMETHODIMP CLowerStrClassFactory::LockServer(BOOL fLock)
{
   if (fLock)
      g_cLock++;
   else
      g_cLock--;
   return S_OK;
}
```

The LockServer first checks fLock to see whether it is true; if yes, the g_cLock will be increased, otherwise, the g_clock will be decreased. This LockServer method will be invoked by the client application.

In the following section, a set of examples will be demonstrated to further illustrate the concept. First, an in-proc server will be created and used. Then this in-proc server will be built as an out-proc server and used. After that, ATL will be used to create this in-proc server.

Create and Use an In-Proc Server

The server(lst31.dll) is illustrated in Figure 3.5. For the complete project, please refer to the lst31 directory on the CD.

Figure 3.5. lst31.dll COM server

There is one interface ILowerStr exposed by the CLowerStr object. There is only one method called Lower in this interface.

virtual STDMETHODIMP Lower(char *lpInput, char**lpOutput) = 0;

This method accepts an input and converts the input string to lowercase and then returns the input string to the caller.

In particular, lpInput in Lower method indicates the input string.

lpOutput indicates the returned string that converts the strInput to uppercase.

Create lst31.dll COM Server

The following steps demonstrate how we implement this in-proc server.

1. 1. Generate two GUIDs in DEFINE_GUID format, one for IID, the other for CLSID by using guidgen.exe. Replace the <<name>> in the code generated by guidgen.exe by IID of the interface and the CLSID of the object class.
2. The CLSID and IID is a universal unique ID (UUID). CLSID stands for class identifier, whereas IID is for interface identifier. The use of this unique ID avoids the possibility of a naming collision among COM objects and interfaces.
3. COM clients use these unique identifiers at runtime to locate the object and its interfaces. COM library uses these IDs to locate the COM server module path in the registry.
4. UUID can be obtained through uuidcreate RPC function. There are tools that directly or indirectly use this function to generate the UUID, such as guidgen.exe, and uuidgen.exe.
5. guidgen.exe is a window-based application and uuidgen.exe is a console based application. They both are contained in Microsoft Visual C++. uuidgen.exe only generates a UUID in a registry format, whereas guidgen.exe generates four formats as shown in Figure 3.6.

Figure 3.6. GUID Formats provided by guidgen.exe.

1. Format 2: "DEFINE_GUID(...)" is used to identify a CLSID or IID. For example, a CLSID is defined in

```
DEFINE_GUID(CLSID_CLowerStr, 0x4f126d90, 0x1319, 0x11d0, 0xa6,



[icc]0xac, 0x0, 0xaa, 0x0, 0x60, 0x25, 0x53);
```

1. Format 4: "Registry Format" is used to build registry entry for the COM server.

```
REGEDIT

; CUpperStr Server Registration

HKEY_CLASSES_ROOT\CLSID\{4F126D90-1319-11d0-A6AC-00AA00602553} =

[icc]CLowerStr Object

HKEY_CLASSES_ROOT\CLSID\{4F126D90-1319-11d0-A6AC-00AA00602553}

[icc]\InprocServer32 = d:\areview\ch3\lst31\debug\lst31.dll
```

1. Format 1: IMPLEMENT_OLECREATE(...) and Format 3: static const struct GUID={...} are not used as often as Format 1 and 4.
2. By convention, symbolic constants are used to identify a specific CLSID or IID. It is in the form of CLSID_<*class name*> or IID_<*interface name*>.

Note

I've used d: for my drive letter, but you should substitute the letter of the hard drive you installed the code to.

---

1. For example, Listing 3.8 demonstrates how to use this format.

Listing 3.8. CLSID and IID.

```
// {4F126D90-1319-11d0-A6AC-00AA00602553}
DEFINE_GUID(CLSID_CLowerStr, 0x4f126d90, 0x1319, 0x11d0, 0xa6, 0xac, 0x0, 0xaa,


 // {4F126D91-1319-11d0-A6AC-00AA00602553}
DEFINE_GUID(IID_ILowerStr, 0x4f126d91, 0x1319, 0x11d0, 0xa6, 0xac, 0x0, 0xaa,
[icc[0x0, 0x60, 0x25, 0x53);
```

1. For the interfaces defined in the COM, such as IUnknown, the IIDs are predefined, because every interface needs to have a IID associated with it.
2. Listing 3.9 demonstrates the IID_ILowerStr, and CLSID_CLowerStr for the CLowerStr object.

Listing 3.9. IID_ILowerStr and CLSID_CLowerStr

```
    // {4F126D90-1319-11d0-A6AC-00AA00602553}
    DEFINE_GUID(CLSID_CLowerStr, 0x4f126d90, 0x1319, 0x11d0, 0xa6,
    [icc]0xac, 0x0, 0xaa, 0x0, 0x60, 0x25, 0x53);
    // {4F126D91-1319-11d0-A6AC-00AA00602553}
    DEFINE_GUID(IID_ILowerStr, 0x4f126d91, 0x1319, 0x11d0, 0xa6,
    [[icc]0xac, 0x0, 0xaa, 0x0, 0x60, 0x25, 0x53);
```

1. 2. Define the interface ILowerStr
2. Listing 3.10 demonstrates the ILowerStr interface definition. Every interface in COM object requires inheritance from IUnknown to provide the interface navigation and reference counting capability.

Listing 3.10. ILowerStr Interface Definition

```
    class ILowerStr    : public IUnknown
    {
    public:
        virtual STDMETHODIMP  Lower(char *lpInput, char**lpOutput) = 0;
    };
```

1. 3. ILowerStr interface implementation

2. Listing 3.11 demonstrates the ILowerStr interface implementation.

Listing 3.11. ILowerStr interface implementation.

```
class CLowerStr : public ILowerStr
{
public:
    STDMETHODIMP QueryInterface(REFIID iid, LPVOID *ppv);
    STDMETHODIMP_(ULONG) AddRef();
    STDMETHODIMP_(ULONG) Release();
    STDMETHODIMP Lower(char *lpString, char**lpOutput);
    CLowerStr();
    ~CLowerStr();
private:
    DWORD m_dwRef;
};
STDMETHODIMP CLowerStr::QueryInterface(REFIID iid, void **ppv)
{
    HRESULT hr;
    *ppv = NULL;
    if((iid == IID_IUnknown) || (iid == IID_ILowerStr) )
    {
        *ppv = (ILowerStr *)this;
        AddRef();
        hr = S_OK;
    }
    else
    {
        *ppv = NULL;
        hr = E_NOINTERFACE;
    }
    return hr;
```

```
}
STDMETHODIMP_(ULONG) CLowerStr::AddRef()
{
    m_dwRef++;
    return m_dwRef;
}
STDMETHODIMP_(ULONG) CLowerStr::Release()
{
    m_dwRef--;
    if(m_dwRef == 0)
        delete this;
    return m_dwRef;
}
STDMETHODIMP  CLowerStr::Lower(char *lpInput, char **pOutput)
{
    int i;
    *pOutput = new char[strlen(lpInput) + 1];
    for (i=0; i< strlen(lpInput); i++)
    {
        if(isupper(*(lpInput+i)))
            *(*pOutput+i) = _tolower(*(lpInput + i));
        else
            *(*pOutput+i) = *(lpInput + i);
    }
    *(*pOutput+i) = '\0';
    return S_OK;
}
CLowerStr::CLowerStr()
{
    m_dwRef = 1;
```

```
}

CLowerStr::~CLowerStr()

{

}
```

1. CLowerStr is the implementation of the ILowerStr interface. In the CLowerStr::QueryInterface(...) method, the AddRef() is invoked whenever an interface is successfully queried.
2. AddRef() and Release() are the most standard implementation.
3. Method Lower(...) accepts the input and converts the input string to lowercase.
4. In the CLowerStr constructor, m_dwRef is initialized to 1 because this object is successfully instantiated by the client application.
5. 4. Implement IClassFactory interface.
6. Listing 3.12 demonstrates implementation of the class factory object for the CLowerStr object. This class factory object implements two methods (CreateInstance, and LockServer) defined in the IClassFactory interface, and three methods(QueryInterface, AddRef, and Release) defined in the IUnknown interface. Because IClassFactory inherits from IUnknown interface.

Listing 3.12. Class Factory implementation.

```
class CLowerStrClassFactory:public IClassFactory

{

  protected:

    DWORD m_dwRef;

  public:

   CLowerStrClassFactory(void);

   ~CLowerStrClassFactory(void);

//IUnknown

   STDMETHODIMP QueryInterface (REFIID iid ,void **ppv);

   STDMETHODIMP_ (ULONG) AddRef(void);

   STDMETHODIMP_ (ULONG) Release(void);

   STDMETHODIMP CreateInstance(IUnknown *punkOuter,REFIID

[icc]iid,void **ppv);

   STDMETHODIMP LockServer(BOOL);

};

 CLowerStrClassFactory::CLowerStrClassFactory()

{

   m_dwRef=1;

}
```

```
CLowerStrClassFactory::~CLowerStrClassFactory()
{
}
STDMETHODIMP CLowerStrClassFactory::QueryInterface (REFIID
[icc]iid,void **ppv)
  HRESULT hr;
  *ppv = NULL;
  if (IID_IUnknown== iid || IID_IClassFactory== iid)
  {
      *ppv=this;
      AddRef();
      hr = S_OK;
  }
  else
  {
      *ppv = NULL;
      hr = E_NOINTERFACE;
  }
  return hr;
}
STDMETHODIMP_(ULONG) CLowerStrClassFactory::AddRef(void)
{
   return m_dwRef++;
}
STDMETHODIMP_(ULONG) CLowerStrClassFactory::Release(void)
{
    m_dwRef--;
    if(m_dwRef == 0)
        delete this;
    return m_dwRef;
```

```
}
STDMETHODIMP CLowerStrClassFactory::CreateInstance (IUnknown
[icc]*pUnkOuter,REFIID
[icc]iid,void **ppv)
   HRESULT hr;
   CLowerStr *pObj;
   *ppv = NULL;
   pObj = new CLowerStr;
   if (pObj)
   {
        hr=pObj->QueryInterface(iid,ppv);
        pObj->Release();
   }
   else
   {
       hr = E_OUTOFMEMORY;
        *ppv = NULL;
   }
   return hr;
}
STDMETHODIMP CLowerStrClassFactory::LockServer(BOOL fLock)
{
   if (fLock)
      g_cLock++;
   else
      g_cLock--;
   return S_OK;
}
```

1. The implementation of AddRef and Release are the same as in CLowerStr in Listing 3.11. QueryInterface is almost the same except the interface exposed by the CLowerStr is different from CLowerStrClassFactory. CLowerStr inherits from two interfaces, IUnknown and ILowerStr, whereas CLowerStrClassFactory inherits IUnknown and IClassFactory.

2. 5 Define and implement export functions from Lst31 COM server.
3. Because lst31.dll is an in-proc server, functions need to be exported to be accessed by the client application. For every in-proc server, the DllGetClassObject function needs to be exported so that COM library can access this function to create an instance of the COM object.
4. Listing 3.13 demonstrates the module-definition (.DEF) file provided for Lst31 COM server.

Listing 3.13. Lst31 COM server .DEF FileLIBRARY CLowerSTR.

```
EXPORTS

    DllGetClassObject  @1

    DllCanUnloadNow    @2
```

1. Exporting DllGetClassObject is mandatory for every in-proc server. It is the function invoked by the COM library to create an instance of the COM object.
2. Listing 3.14 demonstrates the implementation of the DllGetClassObject function.

Listing 3.14. lst31.dll's DllGetClassObject's implementation.

```
long g_cLock = -1;

long g_cObj = 0;

STDAPI DllGetClassObject (REFCLSID rclsid,REFIID riid,void **ppv)

{

    HRESULT hr;

    CLowerStrClassFactory *pObj;

    if (CLSID_CLowerStr!= rclsid)

        return ResultFromScode(E_FAIL);

    pObj = new CLowerStrClassFactory();

    if (!pObj)

        return ResultFromScode (E_OUTOFMEMORY);

    hr= pObj->QueryInterface(riid,ppv);

    if (FAILED(hr))

        delete pObj;

    return hr;

}

STDAPI DllCanUnloadNow (void)

{

  SCODE sc;
```

```
    sc=(0L==g_cObj && 0L==g_cLock)? S_OK : S_FALSE;

    return ResultFromScode (sc);

}
```

1. DllGetClassObject returns the interface to the IClassFactory. This function has three parameters, rclsid is the input parameter, referring to the CLSID of the class object. iid is the input parameter that is the interface ID which the caller uses to communicate with the class object. In most cases, it is IID_IClassFactory. ppv is the return value; ppv is an indirect pointer to the IClassFactory interface of the class factory object.
2. 6. Build lst31.dll.
3. 7. Register Lst31 COM server by providing .REG file.
4. Before a COM server can be used, the proper information such as CLSID and the full path of the DLL has to be stored in the registry under the HKEY_CLASSES_ROOT\CLSID. This is required for every COM server because the registry information will be accessed by the COM library to access the COM server location so that COM library can access the exported functions such as DllGetClassObject from the COM server.
5. Listing 3.15 demonstrates the lst31.REG file for Lst31 COM server. Run regedit /s lst31.reg to register lst31.dll.

Listing 3.15. Lst31 COM server .REG file.

```
REGEDIT

; CUpperStr Server Registration

HKEY_CLASSES_ROOT\CLSID\{4F126D90-1319-11d0-A6AC-00AA00602553}

[icc] = CLowerStr Object

HKEY_CLASSES_ROOT\CLSID\{4F126D90-1319-11d0-A6AC-00AA00602553}

[icc]\InprocServer32 = d:\areview\ch3\lst31\debug\lst31.dll
```

1. For every COM server, all the information has to be stored under the string representation of the CLSID, which is an immediate subkey of the HKEY_CLASSES_ROOT\CLSID.
2. The string representation of the CLSID is in the CLSID's registry format, denoted as {CLSID}. The value associated with the {CLSID} is the description of the COM object. The value associated with the InProcServer32 subkey is the full path to the 32 bit in-proc server. InProcServer32 subkey is defined by OLE to indicate the full path to the 32-bit in-proc server. There are other subkeys, such as Control and LocalServer32; subkeys are defined by OLE to serve their different purposes.
3. Control subkey with no value indicates that the COM server is an OLE control. The value associated with LocalServer32 subkey indicates the full path to the local server.
4. For more information on this, please refer to Chapter 7 in *Windows NT Registry Guide* by Addison-Wesley.
5. For complete project, see lst31 directory in CD.

Use lst31.dll COM Server

The following example will demonstrate how to use the COM server just created (lst31.dll) in a console application. This console application is called lst31use.exe.

Note

COM server is an binary reusable component. It can be used in any applications such as Visual C++, and Visual Basic.

The following steps illustrate how to use lst31.dll.

1. 1. Initialize the COM library by calling CoInitialize(NULL). OleInitialize can be called instead of CoInitialize, because

OleInitialize initializes OLE library, which is a superset of the COM library.

2. 2. Create an instance of the CLowerStr by invoking

```
CoCreateInstance(CLSID_CLowerStr 0, CLSCTX_INPROC_SERVER, IID_

[icc]ILowerStr, (void**)&pLoweStr)
```

1. CLSID_CLowerStr specifies the CLSID of the CLowerStr object.
2. 0 indicates that the object is not created as part of an aggregate.
3. CLSCTX_INPROC_SERVER is one type of CLSCTX. It indicates that lst31.dll is an in-proc server. It will run in the same address space as the lst31use.exe. Listing 3.16 enumerates CLSCTX.

Listing 3.16. CLSCTX enumeration.

```
typedef enum tagCLSCTX
{
    CLSCTX_INPROC_SERVER   = 1,
    CLSCTX_INPROC_HANDLER  = 2,
    CLSCTX_LOCAL_SERVER    = 4
   CLSCTX_REMOTE_SERVER = 16
} CLSCTX;
```

1. CLSCTX_INPROC_HANDLER indicates that the COM server is an in-process handler.
2. CLSCTX_LOCAL_SERVER indicates that the COM server runs on the same machine as the client but in a different process.
3. CLSCTX_REMOTE_SERVER indicates that the COM server runs on a different machine from the client.
4. IID_ILowerStr refers to the interface to communicate with Lst31 COM server.
5. COM also predefines

```
#define CLSCTX_SERVER (CLSCTX_INPROC_SERVER| CLSCTX_LOCAL_

[icc]SERVER| CLSCTX_REMOTE_SERVER)

#define CLSCTX_ALL (CLSCTX_INPROC_HANDLER | CLSCTX_SERVER)
```

1. pLowerStr points to the location of the IID_ILowerStr interface pointer.
2. 3. Check the returned value from CoCreateInstance in Step 2.
3. For a robust system, I strongly suggest checking the returned value.
4. 4, Invoke Lower method

```
    char *lpOutput;
```

```
    pILowerStr->Lower("hEllo World", &lpOutput);

    printf("the output is %s\n", lpOutput);
```

1. 5. Release ILowerStr interface.

```
pLowerStr->Release();
```

1. The pLowerStr needs to be released because it is no longer needed.
2. 6 Invoke CoUninitialize() to uninitialize the COM library so that all COM resources and RPC connections can be released.

Note

OleUninitialize can be invoked to uninitialize OLE library. Because OLE library is a superset of COM library.

```
CoUnInitialize();
```

1. Listing 3.17 demonstrates the complete program for lst31use application.

Listing 3.17. lst31use program.

```
#include <objbase.h>

#include <initguid.h>

#include <stdio.h>

class ILowerStr    : public IUnknown

{

public:

    virtual STDMETHODIMP Lower(char *lpInput, char**lpOutput) = 0;

};

const CLSID CLSID_CLowerStr =

{0x4f126d90, 0x1319, 0x11d0, {0xa6, 0xac, 0x0, 0xaa, 0x0, 0x60,

[icc]0x25, 0x53}};

const CLSID IID_ILowerStr =

{0x4f126d91, 0x1319, 0x11d0, {0xa6, 0xac, 0x0, 0xaa, 0x0, 0x60,

[icc]0x25, 0x53}};

void main()

{

    HRESULT hr;
```

```
 ILowerStr  *pILowerStr;
hr = CoInitialize(NULL);
if(FAILED(hr))
{
    printf("CoInitialize failed[0x%x]\n", hr);
    exit(1);
}
//Create an instance of the COM object
hr = CoCreateInstance(CLSID_CLowerStr, NULL, CLSCTX_INPROC_SERVER,
     IID_ILowerStr,(void**) &pILowerStr);
if(FAILED(hr))
{
    printf("CoCreateInstance failed[0x%x]\n", hr);
    if(hr == REGDB_E_CLASSNOTREG)
        printf("please register the class\n");
    exit(1);
}
char *lpOutput;
pILowerStr->Lower("hEllo World", &lpOutput);
printf("the output is %s\n", lpOutput);
pILowerStr->Release();
CoUninitialize();
}
```

1. Before lst31.dll can be invoked, the interface ILowerStr needs to be included and so does the CLSID and IID declared by the COM server.

```
lst31.dll
```

1. This complete project is under the lst31use directory contained on the CD.

Create lst31 COM Server New Version

Assumed some new requirements coming up for lst31.dll COM server, a new version needs to be released. In a lot of circumstance, the client application that uses the dll needs to be recompiled if the library is statically linked. But the COM client using the old version of lst31.dll can use the new version of lst31.dll without any changes to source code and compilation.

Figure 3.7 illustrates the functionality provided by the new version of lst31.dll.

Figure 3.7. New version of lst31.dll.

There are two interfaces exposed by the lst31.dll. ILowerStr is the old interface, IHelpLowerStr is the new interface. This new interface includes one method HelpLower() which provides the help information, here for simplicity, S_OK will be returned.

The following steps demonstrate how to create the new lst31.dll COM server.

1. 1. Generate a new IID for new interface IHelpLowerStr

```
// {9EC1CA01-133F-11d0-A6AD-00AA00602553}
DEFINE_GUID(IID_IHelpLowerStr, 0x9ec1ca01, 0x133f, 0x11d0, 0xa6,
[icc]0xad, 0x0, 0xaa, 0x0, 0x60, 0x25, 0x53);
```

1. 2. Define the new interface IHelpLowerStr

```
class IHelpLowerStr: public IUnknown
{
public:
    virtual STDMETHODIMP HelpLower() = 0;
};
```

1. 3. Implement IHelpLowerStr by modifying CLowerStr class. The changes are highlighted in bold font in Listing 3.18.

Listing 3.18. New CLowerStr class.

```
class IHelpLowerStr : public IUnknown
{
public:
    virtual STDMETHODIMP HelpLower()=0;
};
class CLowerStr : public ILowerStr , public IHelpLowerStr
{
public:
    STDMETHODIMP QueryInterface(REFIID iid, LPVOID *ppv);
    STDMETHODIMP_(ULONG) AddRef();
    STDMETHODIMP_(ULONG) Release();
    STDMETHODIMP Lower(char *lpString, char**lpOutput);
```

```
    STDMETHODIMP HelpLower();
    CLowerStr();
    ~CLowerStr();
private:
    DWORD m_dwRef;
};
STDMETHODIMP CLowerStr::QueryInterface(REFIID iid, void **ppv)
{
    HRESULT hr;
    *ppv = NULL;
    if((iid == IID_IUnknown) || (iid == IID_ILowerStr) )
    {
        *ppv = (ILowerStr *)this;
        AddRef();
        hr = S_OK;
    }
    else if(iid == IID_IHelpLowerStr)
    {
        *ppv = (IHelpLowerStr *) this;
        AddRef();
        hr = S_OK;
    }
    else
    {
        *ppv = NULL;
        hr = E_NOINTERFACE;
    }
    return hr;
}
STDMETHODIMP CLowerStr::HelpLower()
```

```
        {
            return S_OK;
        }
```

1. Because a new interface (IHelpLowerStr) is exposed by the CLowerStr object, QueryInterface needs to check IHelpLowerStr interface, too.
2. 4. IUnknown] Rebuild lst311 project.
3. 5. Create Registry setting

Following is the listing for the lst311.reg file.

```
REGEDIT

; CUpperStr Server Registration

HKEY_CLASSES_ROOT\CLSID\{4F126D90-1319-11d0-A6AC-00AA00602553} = CLowerStr Object

HKEY_CLASSES_ROOT\CLSID\{4F126D90-1319-11d0-A6AC-00AA00602553}

[icc]\InprocServer32 = d:\areview\ch3\lst311\debug\lst311.dll
```

Run lst31use.exe. It still works! But the value associated with the InprocServer32 has been changed, it is d:\areview\ch3\lst311\debug\lst311.dll now. The reason is that from the client perspective, nothing really cares what dll name is associated with the COM server. The client side is only interested in the CLSID exposed by the COM server. It is the COM library's role to locate the COM server and instantiate the COM object. The library also returns the interface pointer requested by the client application.

The complete project is located under the lst311 directory on the CD.

Create and Use an out-proc Server

To illustrate how to create an out-proc server, changes will be made to lst31.dll so that it is an out-proc server.

The out-proc server runs in a separate address space from the client application; all interface calls must be marshaled across process (data gets copied). It is more reliable compared with in-proc server because the corruption of the out-proc server will not influence the process space of the caller application, whereas in-proc server will destroy the caller's process space. The local out-proc server should be created when an application supports automation. A remote out-proc server should be used to take advantage of resources on another machine.

The following steps illustrate how to create Lst32 out-proc server.

1. [1] Create the custom interface by using Interface Definition Language(IDL).
2. Listing 3.19 demonstrates the lower.idl file for ILowerStr interface. Because all interface calls must be marshaled for out-proc server, ILowerStr interface must be marshaled.

Listing 3.19. lower.idl for ILowerStr interface

```
        [
            object,
            uuid(4F126D91-1319-11d0-A6AC-00AA00602553),
            pointer_default(unique)
        ]
```

```
interface ILowerStr: IUnknown
{
    import "oaidl.idl";
    HRESULT Lower([in,string] LPSTR inString, [out, string] LPSTR *outString);
}
```

1. An IDL file specifies the contract between the client and server using IDL.
2. It consists of two parts, the interface header and the interface body.
3. Table 3.1 illustrates the attributes for IDL. For more information on the IDL, please refer to the Microsoft RPC document.

Table 3.1. IDL attributes.

Keyword Meaning object COM interface object. uuida universal unique identifier associated with particular interface usage uuid(80FA6EE2-0120-11d0-A6A0-00AA00602553), dual Dual interface, which inherits from IDispatch, IUnknowndual, import Imports idl file of the base interface import oaidl.idl out Output parameter [out, retval]BSTR *pBSTR in input parameter in, out Data is sent to the object initialized and will be changed before sending it back. retval Designates the parameter that receives the return value helpstring Sets the help stringhelpstring("test only") pointer_default Specifies the default pointer attribute unique pointer Attribute,designates a pointer as a full pointer ref pointer attribute, Identifies a reference pointer.

1. 2. Create makefile for lower.idl.
2. Listing 3.20 demonstrates the makefile for compiling x.

Listing 3.20. Makefile for lower.idl.# Change CPU to MIPS or ALPHA for compiling on those platforms

```
CPU=i386
TARGETOS=BOTH
!include <win32.mak>
all:  lower.dll
.cxx.obj:
    $(cc) $(cflags) $(cvarsmt) $<
.c.obj:
    $(cc)  $(cflags) $(cvarsmt) $<
#the files that make up the dll
lower_i.obj : lower_i.c
lower_p.obj : lower_p.c lower.h
dlldata.obj : dlldata.c
    $(cc)  $(cflags) $(cvarsmt) -DREGISTER_PROXY_DLL dlldata.c
# run midl to produce the header files and the proxy file
lower.h lower_p.c lower_i.c dlldata.c: lower.idl
```

```
                midl /ms_ext /c_ext lower.idl

lower.dll: lower_p.obj lower_i.obj dlldata.obj lower.def

    $(link)             \

    -dll                \

    -entry:_DllMainCRTStartup$(DLLENTRY) \

    -DEF:lower.def    \

    -out:lower.dll    \

    lower_p.obj lower_i.obj  dlldata.obj rpcrt4.lib $(olelibs)

# Clean up everything

cleanall: clean

        @-del *.dll 2>nul

# Clean up everything but the .EXEs

clean:

        @-del *.obj 2>nul

        @-del dlldata.c 2>nul

        @-del *.h 2>nul

        @-del lower_?.* 2>nul

        @-del *.exp 2>nul

        @-del *.lib 2>nul

        @-del *.ilk 2>nul

        @-del *.pdb 2>nul

        @-del *.res 2>nul
```

1. midl in Listing 3.20 stands for Microsoft IDL compiler. It takes the lower.idl file and generates the following files:
2. lower_p.c contains proxy/stub code
3. lower_i.c contains the actual definition of IIDs and CLSIDs.
4. lower.h contains the definition for the interface
5. dlldata.c regenerated by MIDL compiler on every idl file.
6. The midl switch /m_ext will support Microsoft extensions to DCE IDL. These extensions include: Interface definition for OLE objects, multiple interfaces, enumeration, cpp_quote (quoted_string) and wide character types(wchar_t), and so on.
7. Let's say that if the IDL file includes

```
typedef enum

{
```

```
A=1,

B,

C

} BAD_ENUM;
```

1. /m_ext switch needs to be turned on.
2. /c_ext switch enables the use of C-language extensions in the IDL file. For instance, if // is used to comment the code, /c_ext switch needs to be on.
3. 2. Lower.Def

```
LIBRARY      LOWER

DESCRIPTION  'Proxy/Stub DLL for ILowerStr interfaces'

EXPORTS

        DllGetClassObject         PRIVATE

        DllCanUnloadNow           PRIVATE

        DllRegisterServer         PRIVATE

        DllUnregisterServer       PRIVATE
```

1. 3. Build proxystub dll.
2. 4. Register custom interface.
3. run command: regsvr32 lower.dll.
4. *The* ILowerStr interface information will be written to the system registry under HKEY_CLASSES_ROOT\Interface
5. For the complete lower.dll project, please refer to lst32\proxy.
6. 5. Create a main entry point for Lst32.exe.

Listing 3.21 demonstrates a main program for lst32.exe.

Listing 3.21. Lst32.exe main program.

```
HRESULT RegisterClassFactory()

{

    HRESULT hr;

    CLowerStrClassFactory *pClassFactory;

    pClassFactory = new CLowerStrClassFactory;

    if (pClassFactory != 0)

    {

        hr = CoRegisterClassObject(CLSID_CLowerStr,

                                   (IUnknown *) pClassFactory,

                                CLSCTX_LOCAL_SERVER,
```

```
                                        REGCLS_SINGLEUSE,
                                        &g_dwRegister);
            pClassFactory->Release();
            hr = S_OK;
        }
        else
        {
            hr = E_OUTOFMEMORY;
        }
        return hr;
    }
```

1. CoRegisterClassObject is the function that needs to be called on startup. It registers OLE so that other applications can connect to this class object. There are 5 parameters in the function. In particular, REGCLS_SINGLEUSE indicates the types of connection to the class object. If an application has connected to the class object via CoGetClassObject, no other application can connect to it.
2. There are other two connection types, they are REGCLS_MULTIPEUSE and REGCLS_MULTI_SEPARATE. REGCLS_MULTIPEUSE indicates that multiple applications can connect to the class object via CoGetClassObject, whereas REGCLS_MULTI_SEPARATE indicates that the application has separate control over the each copy of the class object context.
3. g_dwRegister is a returned value, identifying the class object registered. It will be used in CoRevokeClassObject function call.

```
    HRESULT RevokeClassFactory()
    {
        HRESULT hr;
        hr = CoRevokeClassObject(g_dwRegister);
        return hr;
    }
```

1. CoRevokeClassObject function indicates that the class object, previously registered with OLE via CoRegisterClassObject function is no longer available for use.

```
    void main(int argc, char *argv[])
    {
        HRESULT hr = S_OK;
        int i;
        BOOL bRegisterServer = FALSE;
        BOOL bUnregisterServer = FALSE;
        MSG msg;
```

```
    for (i = 1; i < argc; i++)
    {
        if (_stricmp( argv[i], "/REGSERVER" ) == 0) {
            bRegisterServer = TRUE;
        }
        else if (_stricmp( argv[i], "/UNREGSERVER" ) == 0) {
            bUnregisterServer = TRUE;
        }
    }
    if(bRegisterServer)
    {
        RegisterLocalServer(CLSID_CLowerStr);
        return;
    }
    if(bUnregisterServer)
    {
        UnregisterLocalServer(CLSID_CLowerStr);
        return;
    }
    hr = CoInitialize(NULL);
    if (FAILED(hr)) {
        printf("CoInitialize failed [0x%x].\n", hr);
        return;
    }
    hr = RegisterClassFactory();
    if (SUCCEEDED(hr))
    {
        printf("Waiting for client to connect...\n");
        while (GetMessage(&msg, NULL, 0, 0))
        {
```

```
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
        RevokeClassFactory();
    }
    else
    {
        printf("Failed to register class factory [0x%x].\n", hr);
    }
    CoUninitialize();
    return;
}
HRESULT RegisterLocalServer(REFCLSID rclsid)
{
    HRESULT hr;
    LONG lError;
    HKEY hKeyCLSID;
    HKEY hKeyClassID;
    HKEY hKey;              // current key
    DWORD dwDisposition;
    char szServer[MAX_PATH];
    char szClassID[39];
    ULONG ulLength;
    ulLength = GetModuleFileNameA(0, szServer, sizeof(szServer));
    if (ulLength == 0)
    {
        hr = HRESULT_FROM_WIN32(GetLastError());
        return hr;
    }
    //create the CLSID key
```

```
    lError = RegCreateKeyExA(
            HKEY_CLASSES_ROOT,
            "CLSID",
            0,
            "REG_SZ",
            REG_OPTION_NON_VOLATILE,
            KEY_ALL_ACCESS,
            0,
            &hKeyCLSID,
            &dwDisposition);
    if (!lError) {
        //convert the class ID to a registry key name.
        sprintf(szClassID,
                "{%08lX-%04X-%04X-%02X%02X-%02X%02X%02X%02X%02X%02X}",
                rclsid.Data1, rclsid.Data2, rclsid.Data3,
                rclsid.Data4[0], rclsid.Data4[1],
                rclsid.Data4[2], rclsid.Data4[3],
                rclsid.Data4[4], rclsid.Data4[5],
                rclsid.Data4[6], rclsid.Data4[7]);
        //create key for the server class
        lError = RegCreateKeyExA(hKeyCLSID,
                                 szClassID,
                                 0,
                                 "REG_SZ",
                                 REG_OPTION_NON_VOLATILE,
                                 KEY_ALL_ACCESS,
                                 0,
                                 &hKeyClassID,
                                 &dwDisposition);
        if (!lError) {
```

```
            //create LocalServer32 key.
            lError = RegCreateKeyExA(hKeyClassID,
                                     "LocalServer32",
                                     0,
                                     "REG_SZ",
                                     REG_OPTION_NON_VOLATILE,
                                     KEY_ALL_ACCESS,
                                     0,
                                     &hKey,
                                     &dwDisposition);
            if (!lError) {
                //Set the server name.
                lError = RegSetValueExA(hKey,
                                        "",
                                        0,
                                        REG_SZ,
                                        (const unsigned char *)szServer,
                                        strlen(szServer) + 1);
                RegFlushKey(hKey);
                RegCloseKey(hKey);
            }
            RegCloseKey(hKeyClassID);
        }
        RegCloseKey(hKeyCLSID);
    }
    if (!lError)
        hr = S_OK;
    else
        hr = HRESULT_FROM_WIN32(lError);
    return hr;
```

```
}
HRESULT UnregisterLocalServer(REFCLSID rclsid)
{
    HRESULT hr;
    HKEY hKeyCLSID;
    HKEY hKeyClassID;
    long lError;
    char szClassID[39];
    //open the CLSID key
    lError = RegOpenKeyExA(
                HKEY_CLASSES_ROOT,
                "CLSID",
                0,
                KEY_ALL_ACCESS,
                &hKeyCLSID);
    if (!lError) {
        //convert the class ID to a registry key name.
        sprintf(szClassID,
                "{%08lX-%04X-%04X-%02X%02X-%02X%02X%02X%02X%02X%02X}",
                rclsid.Data1, rclsid.Data2, rclsid.Data3,
                rclsid.Data4[0], rclsid.Data4[1],
                rclsid.Data4[2], rclsid.Data4[3],
                rclsid.Data4[4], rclsid.Data4[5],
                rclsid.Data4[6], rclsid.Data4[7]);
        //open registry key for class ID string
        lError = RegOpenKeyExA(
                hKeyCLSID,
                szClassID,
                0,
                KEY_ALL_ACCESS,
```

```
                    &hKeyClassID);
            if (!lError) {
                //delete LocalServer32 key.
                lError = RegDeleteKeyA(hKeyClassID, "LocalServer32");
                RegCloseKey(hKeyClassID);
            }
            lError = RegDeleteKeyA(hKeyCLSID, szClassID);
            RegCloseKey(hKeyCLSID);
        }
        if (!lError)
            hr = S_OK;
        else
            hr = HRESULT_FROM_WIN32(lError);
        return hr;
    }
```

1. lst32.exe provides the self registration features. Self registration means that the COM server can register itself. An in-proc server registers by providing two entry points by the in-proc server, they are

```
HRESULT DllRegisterServer(void)
```

```
HRESULT DllUnRegisterServer(void);
```

1. DllRegisterServer entry point adds or updates registry information for all the classes implemented by the in-proc server. The DllUnRegisterServer entry point removes all the information for the in-proc server from the registry.
2. For an out-proc server, there is no way to publish well-known entry points. The self registration for out-proc server is supported by using special command-line flags. The command-line flags are

```
/regserver argument and  /unregister argument.
```

1. /regserver argument should add the registry information for all classes implemented by the out-proc server and then exit. /unregister argument should do all the necessary uninstallation and then exit.
2. lst32.exe supports the /regserver and /unregserver argument.
3. 6. Register lst32.exe by running lst32 /regserver.
4. Lst32.exe out-proc server needs to be registered before being used. Without proper information in the registry, the out-proc server will not be seen by any other COM applications or COM library.
5. Because lst32.exe supports self registration, lst32.exe can be registered by invoking lst32 /regserver.

Use lst32.exe COM Server

A COM client will be created to use this out-proc (lst32.exe) COM server. Listing 3.22 demonstrates how to use lst32.exe.

Listing 3.22. Lst32use.exe program.

```
#include <windows.h>
#include <stdio.h>
#include <olectl.h>
#include <initguid.h>
#include <olectlid.h>
// the class ID of the server exe
// {4F126D90-1319-11d0-A6AC-00AA00602553}
const CLSID CLSID_CLowerStr =
{0x4f126d90, 0x1319, 0x11d0, {0xa6, 0xac, 0x0, 0xaa, 0x0, 0x60, 0x25, 0x53}};
 // {4F126D91-1319-11d0-A6AC-00AA00602553}
const CLSID IID_ILowerStr =
{0x4f126d91, 0x1319, 0x11d0, {0xa6, 0xac, 0x0, 0xaa, 0x0, 0x60, 0x25, 0x53}};
class ILowerStr: public IUnknown
{
public:
    virtual STDMETHODIMP Lower(char* lpInput, char **lpOutput)=0;
};
void __cdecl main(int argc, char *argv[])
{
    ILowerStr *pILowerStr = NULL;
    HRESULT hr;
    hr = CoInitialize(NULL);
    if (FAILED(hr))
    {
        printf("CoInitialize failed [0x%x]\n", hr);
```

```
        exit(1);
    }
    hr = CoCreateInstance(CLSID_CLowerStr, 0, CLSCTX_LOCAL_SERVER,
        IID_ILowerStr,(void**)&pILowerStr);
    if (FAILED(hr))
    {
        printf("CoCreateInstance failed [0x%x]\n", hr);
        if (hr == REGDB_E_CLASSNOTREG)
        {
            printf("Run lst32.exe /REGSERVER to install server program.\n");
        }
        exit(1);
    }
    char *lpOutput;
    pILowerStr->Lower("HELLO", &lpOutput);
    printf("this is it %s\n", lpOutput);
    pILowerStr->Release();
    CoUninitialize();
}
```

From Listing 3.22, no changes need to be made from lst31use.cpp to lst32use.cpp except in the CoCreateInstance activation call. The execution context changes from CLSCTX_INPROC_SERVER to CLSCTX_LOCAL_SERVER.

Create and Use an in-proc COM Server by Using ATL

Active Template Library is an OLE COM AppWizard providing the framework for building COM servers.

From the previous section, a lot of implementation, such as IUnknown and IClassFactory can be reused as noticed. ATL encapsulates these implementations in a template class so that the COM server functionality can be concentrated.

Create lst33.dll COM Server

lst33.dll will be created to illustrate the ATL. lst33.dll provides the same functionality as lst31.dll Version 1.

The following steps demonstrate how to create lst33.dll by using ATL.

1. 1. From the *File* menu, choose *new*, and select *project workspace*.
2. 2. In the new project workspace dialog box, select OLE COM Appwizard. Enter lst33 in the name text box. Click the Create…

button. The dialog box in Figure 3.8 will be displayed.

Figure 3.8. OLE COM Appwizard: Step 1 of 2.

1. 3. In the dialog box shown in Figure 3.8, select the generate IDL only option. Note: This option can be chosen only when MIDL version 3.0 or higher is available.
2. 4. In dialog box shown in Figure 3.8, select the Choose Custom Interface option. Note: Custom interface refers to the fact that all the interfaces are inherited from IUnknown.
3. 5 Click Finish button.

The OLE COM AppWizard will generate the new skeleton project with the following files.

- StdAfx.cpp: source file that includes just the standard include file.
- StdAfx.h: include file for standard system include files.
- lst33.cpp: the DLL initialization code, the same role as in example 1.
- lst33.def: As described before, an in-proc server needs to expose DllGetClassObject so that COM can use this exported function. DllRegisterServer should be exported to support the self-registration function. The wizard will generate all these exported functions in lst33.def, which is equivalent to lst31.def.
- resource.h: include file generated by Microsoft Developer Studio, used by lst33.rc.
- lst33.rc: resource script generated by Microsoft Developer Studio.
- lst33obj.h: object class definition
- lst33obj.cpp: object class implementation.
- lst33.mak: COM server project
- lst33ps.def: a custom interface needs to be registered and export the proper functions as described in lower.def. This wizard generates all these automatically.
- ps.mak: proxystub makefile to compile proxystub dll for the custom interface, the same as created in lower.mak.
- lst33.idl: IDL source file.

This also conforms to the DCOM design, because all of the custom interface has been marshaled.

It will not be difficult to spot where the changes need to be made.

1. 1. There is one interface in this method, the lst33.idl will be changed to add the Upper method.
2. The changes in the skeleton code created by the ATL will be in bold font.

```
interface ILst33 : IUnknown

    {

        import "oaidl.idl";

        HRESULT Upper([in,string] LPSTR inputString, [out, string] LPSTR
*pOutputString);

    };

[2] Implement this method for ILst33 interface, exposed by the lst33 object class by
adding the definition in the lst33obj.h

class CLst33Object :

    public ILst33,

    public CComObjectBase<&CLSID_Lst33>

{

public:
```

```
    CLst33Object() {}
BEGIN_COM_MAP(CLst33Object)
    COM_INTERFACE_ENTRY(ILst33)
END_COM_MAP()
// Use DECLARE_NOT_AGGREGATABLE(CLst33Object) if you don't want your object
// to support aggregation
DECLARE_AGGREGATABLE(CLst33Object)
// ILst33
public:
    STDMETHOD(Lower)( LPSTR bstrInput, LPSTR *pbstrOutput);
};
```

1. 2. The method STDMETHOD(Lower) is added in the object definition.
2. 3. Implement the method in the ILst33 interface by adding the following code in lst33obj.cpp.

```
STDMETHODIMP CLst33Object::Lower(LPSTR lpInput, LPSTR* pOutput)
{
    int i;
    *pOutput = new char[strlen(lpInput) + 1];
    for (i=0; i< strlen(lpInput); i++)
    {
        *(*pOutput+i) = *(lpInput + i)-'A'+'a';
    }
    *(*pOutput+i) = '\0';
    return S_OK;
}
```

1. The implementation of the Lower method is the same as implemented in lst31.dll, and lst32.dll.
2. After the custom interface has been implemented, the following steps need to follow to create this COM server.
3. Run the command midl lst33.idl. This requires MIDL 3.0 or higher. Otherwise, the following error will be generated:

Microsoft MIDL Compiler Version 2.00.0102

```
    Copyright  Microsoft Corp 1991 -1995. All rights reserved.
    Processing .\lst33.idl
    .\lst33.idl(8) : error MIDL2141 : use of this attribute
```

```
[icc]   needs /ms_ext :[object]

   .\lst33.idl(10) : error MIDL2017 : syntax error :expecting an idl

[icc]   attribute near "helpstring"

   .\lst33.idl(10) : error MIDL2018 : cannot recover from earlier

[icc]   syntax errors; aborting compilation
```

1. In order to avoid this error, MIDL 3.0 must be available. It is contained in the Win32 SDK.
2. After lst33.idl is successfully compiled, the following files will be generated:

```
lst33.tlb: type library

lst33_p.c: proxy code

dlldata.c:

lst33_i.c
```

1. b. Build the COM server project (lst33.mak).
2. COM server lst33.dll will be generated.
1. Before this COM server can be used, it needs to be registered first by running regsvr32 lst33.dll. Or the these steps can be followed to register the server after compile.
2. 1. Choose Build|Setting.
3. 2. Select Custom Build tab in the Project Setting dialog box displayed in Figure 3.9.

Figure 3.9. Project Setting—Custom Build.

1. 3. In the Build command list box, enter

```
regsvr32 /s /c "$(TargetPath)"

echo regsvr32 exec. time > $(OutDir)\regsvr32.trg
```

1. 4. In the Output file list box, enter

```
$(OutDir)\regsvr32.trg
```

1. The Microsoft Developer Studio will perform the following Custom Build Step

```
egSvr32: DllRegisterServer in .\Debug\lst33.dll
```

1. after successfully compiling lst33.dll.
2. But this will cause lst33.dll to register every time the project is built. To avoid this, the custom build option can be deleted.

Use Lst33 COM Server

A console application will be created to use the lst33.dll COM server shown in Listing 3.23.

Listing 3.23. Lst33Use.exe program

```
#include <objbase.h>
#include <initguid.h>
#include <stdio.h>
// These equivalent definitions will be from lst33_i.c
#include "..\lst33\lst33.h"
void main()
{
    HRESULT hr;
     ILst33  * m_pILst33; // interface pointer
    CLSID    clsid;
    hr = CoInitialize(NULL);
    hr = CLSIDFromProgID(L"LST33.Lst33Object.1",&clsid);
    hr = CoCreateInstance(clsid,
                          NULL,
                          CLSCTX_INPROC_SERVER,
                          IID_ILst33,
                          (LPVOID*)&m_pILst33);
    if(FAILED(hr))
    {
        printf("can not create Lst33");
        CoUninitialize();
        return;
    }
    char *lpLowerString;
    m_pILst33->Lower("HELLO", & &lpLowerString);
    printf("the return string is %s\n", lpLowerString);
    m_pILst33->Release();
    CoUninitialize();
}
```

Here, CLSIDFromProgID is used to retrieve the CLSID of the class object by providing ProgId for this COM server. No code change is made compared with lst31use.cpp.

Summary

From the previous example, it is not difficult to see that COM supports

Versioning: Provides an interface without COM client's concern.

Network independence: An out-of-process COM server can run locally as the client application or remotely. A COM server can be DLL-based or EXE-based without the COM application doing extra work except informing the COM library about the execution context of the class object.

Language independence: A COM server can be used in C++ applications and any other applications such as Visual Basic. For a basic COM object that only supports IUnknown interface, there is a lot of work to be done so that Visual Basic can use the COM server. For examples on how to use basic COM objects in Visual Basic, please refer to "MFC/COM Objects 6: Using COM Objects from Visual Basic" in MSDN.

In order for Visual Basic application easily to use COM server, the IDispatch interface is defined in COM. For more information on this, please refer to Chapter 4, "Creating OLE Automation Server."

Chapter 4

Creating OLE Automation Server

by Weiying Chen

An OLE Automation server exposes automation objects that have methods and properties as their external interfaces. The methods and properties exposed by the automation objects can be directly accessed by the automation controllers such as Visual Basic or by invoking methods defined in the IDispatch interface.

In this chapter, an automation server will be created through the implementation of the IDispatch interface to illustrate the fundamental OLE automation concept. This automation server will be used in both C++ and Visual Basic.

Then, this automation server will be created through the use of MFC AppWizard (exe), Control Wizard, and the Active Template Library (ATL).

At the end of the chapter, this automation server will be used in Internet Explorer 3.0 and the HTTP Web Server.

Create and Use an Automation Server: lst41.exe

lst41.exe is an automation server implemented without using any wizards or code generation tools. This automation server exposes one method called GetMachineName. GetMachineName has no input parameter. The return value is the name of the computer where the application runs.

To expose the objects, IDispatch interface must be implemented. There are two ways to implement IDispatch interface. One way

is to implement four methods in the IDispatch interface. The other way used in lst41.exe is to expose the objects through OLE automation by using the CreateStdDispatch method. CreateStdDispatch creates a standard implementation of the IDispatch interface through a single function call.

The following steps illustrates how to create lst41.exe.

1. 1. Define the interface using Object Definition Language (ODL).
2. Listing 4.1 highlights the lst41.odl.

Listing 4.1. lst41.odl.

```
[
  uuid(9FBBEDE2-1B40-11d0-88E0-00AA004A7C7B),
  helpstring("Lst41 Type Library"),
  lcid(0x0409),
  version(1.0)
]
library Lst41
{
    importlib("stdole32.tlb");
    [
     odl,
     uuid(9FBBEDE3-1B40-11d0-88E0-00AA004A7C7B),
    ]
    interface ILst41: IUnknown
    {
        BSTR GetMachineName(void);
    }
    [
     uuid(9FBBEDE4-1B40-11d0-88E0-00AA004A7C7B),
    ]
    dispinterface DLst41
    {
      interface ILst41;
```

```
        }
        [
          uuid(9FBBEDE5-1B40-11d0-88E0-00AA004A7C7B),
          helpstring("Lst41")
        ]
        coclass CLst41
        {
          dispinterface DLst41;
          interface ILst41;
        }
    };
```

1. In Listing 4.1,the uuid attribute specifies the UUID of the item.
2. The lcid attribute indicates that the parameter is a locale ID that provides locale information for the international string comparisons and localized member names.
3. The helpstring attribute sets the help string associated with library or interface and so on. This information can be retrieved via the GetDocumentation function in the ITypeLib and ITypeInfo interfaces to retrieve the documentation string.
4. The version attribute specifies a version number. Here, the version number is 1.0.
5. The odl attribute indicates that this interface is an ODL interface.
6. The library statement defines a type library.
7. The importlib directive indicates that stdole32.tlb will be accessible from lst41.tlb.
8. The interface statement defines a interface that is a set of functions. The attribute is described in a function called propget, and propput.
9. The dispinterface statement defines a set of properties and methods that can be called by IDispatch::Invoke. Besides listing a single interface (interface Ilst41) in Listing 4.1, a dispinterface can also be defined by listing the set of methods and properties, for example,
10. **interface** ILst41: IUnknown

```
        {
            BSTR GetMachineName(void);
        }
        [
         uuid(9FBBEDE4-1B40-11d0-88E0-00AA004A7C7B),
        ]
        dispinterface DLst41
```

```
    {
      interface ILst41;
    }
```

1. can be described as

```
dispinterface DLst41
{
    BSTR GetMachineName(void);
}
```

The coclass statement defines the class ID for class name CLst41 and the interfaces supported by CLst41.

1. 2. Create the lst41.tlb type library
2. The type library is used by the automation clients to access the methods and properties exposed by the automation server.
3. Lst41.tlb can be generated by executing the following command:

```
mktyplib -h ilst41.h lst41.odl
```

1. mktyplib is a type library creation tool. It processes lst41.odl scripts and produces ilst41.h header file and a type library lst41.tlb. lst41.tlb can be read by ITypeInfo or ITypeLib interfaces.
2. The ilst41.h header file is shown in Listing 4.2.

Listing 4.2. C++ header file: ilst41.h

```
/* This header file machine-generated by mktyplib.exe */
/* Interface to type library: Lst41 */
#ifndef _Lst41_H_
#define _Lst41_H_
DEFINE_GUID(LIBID_Lst41,0x9FBBEDE2L,0x1B40,0x11D0,0x88,0xE0,0x00,
0xAA,0x00,0x4A,0x7C,0x7B);
#ifndef BEGIN_INTERFACE
#define BEGIN_INTERFACE
#endif
DEFINE_GUID(IID_ILst41,0x9FBBEDE3L,0x1B40,0x11D0,0x88,0xE0,0x00,
0xAA,0x00,0x4A,0x7C,0x7B);
/* Definition of interface: ILst41 */
```

```
#undef INTERFACE
#define INTERFACE ILst41
DECLARE_INTERFACE_(ILst41, IUnknown)
{
BEGIN_INTERFACE
#ifndef NO_BASEINTERFACE_FUNCS
    /* IUnknown methods */
    STDMETHOD(QueryInterface)(THIS_ REFIID riid, LPVOID FAR* ppvObj) PURE;
    STDMETHOD_(ULONG, AddRef)(THIS) PURE;
    STDMETHOD_(ULONG, Release)(THIS) PURE;
#endif
    /* ILst41 methods */
    STDMETHOD_(BSTR, GetMachineName)(THIS) PURE;
};
DEFINE_GUID(DIID_DLst41,0x9FBBEDE4L,0x1B40,0x11D0,0x88,0xE0,0x00,
0xAA,0x00,0x4A,0x7C,0x7B);
/* Definition of dispatch interface: DLst41 */
#undef INTERFACE
#define INTERFACE DLst41
DECLARE_INTERFACE_(DLst41, IDispatch)
{
BEGIN_INTERFACE
#ifndef NO_BASEINTERFACE_FUNCS
    /* IUnknown methods */
    STDMETHOD(QueryInterface)(THIS_ REFIID riid, LPVOID FAR* ppvObj) PURE;
    STDMETHOD_(ULONG, AddRef)(THIS) PURE;
    STDMETHOD_(ULONG, Release)(THIS) PURE;
    /* IDispatch methods */
```

```
    STDMETHOD(GetTypeInfoCount)(THIS_ UINT FAR* pctinfo) PURE;
    STDMETHOD(GetTypeInfo)(
      THIS_
      UINT itinfo,
      LCID lcid,
      ITypeInfo FAR* FAR* pptinfo) PURE;
    STDMETHOD(GetIDsOfNames)(
      THIS_
      REFIID riid,
      OLECHAR FAR* FAR* rgszNames,
      UINT cNames,
      LCID lcid,
      DISPID FAR* rgdispid) PURE;
    STDMETHOD(Invoke)(
      THIS_
      DISPID dispidMember,
      REFIID riid,
      LCID lcid,
      WORD wFlags,
      DISPPARAMS FAR* pdispparams,
      VARIANT FAR* pvarResult,
      EXCEPINFO FAR* pexcepinfo,
      UINT FAR* puArgErr) PURE;
#endif
/* Capable of dispatching all the methods of interface ILst41 */
};
DEFINE_GUID(CLSID_CLst41,0x9FBBEDE5L,0x1B40,0x11D0,0x88,0xE0,0x00,
0xAA,0x00,0x4A,0x7C,0x7B);
```

```
#ifdef __cplusplus

class CLst41;

#endif

#endif
```

1. From Listing 4.2,

```
dispinterface DLst41

    {

      interface ILst41;

    }
```

1. has been expanded into seven methods, three IUnknown methods and four IDispatch methods, which include GetTypeInfoCount, GetTypeInfo, GetIDsOfNames, and Invoke.
2. Method GetTypeInfoCount retrieves the number of type information interfaces provided by an object. If the object provides the type information, pctinfo will be 1, otherwise it will be 0.
3. GetTypeInfo method retrieves a type-information object. This object can be used to get the type information for an interface.
4. GetIDsOfNames method retrieves a DISPID corresponding to the methods and arguments provided.
5. Invoke method accesses the properties and methods given their DISPID.
6. 3. Implement the ILst41 interface. (See Listing 4.3.)

Listing 4.3. ILst41 implementation.
Clst41I and CLst41 class definition

```
#include <objbase.h>

#include "clsid.h"

#include "ilst41.h"

class CLst41;

class CLst41I : public ILst41

{

  public:

STDMETHOD(QueryInterface)(THIS_ REFIID riid, LPVOID* ppvObj);

    STDMETHOD_(ULONG, AddRef)(THIS);

    STDMETHOD_(ULONG, Release)(THIS);

    STDMETHOD_(BSTR, GetMachineName)(THIS);
```

```
    CLst41*    m_pLst41;
};
class CLst41 : public IUnknown
{
 public:
    CLst41();
    ~CLst41();
    static CLst41* Create();
    STDMETHOD(QueryInterface)(REFIID riid, void ** ppv);
    STDMETHOD_(ULONG, AddRef)(void);
    STDMETHOD_(ULONG, Release)(void);
 private:
    ULONG m_refs;            // Reference count.
    IUnknown* m_disp_interface;    // Pointer to the standard dispatch object.
    CLst41I* m_prog_interface;  //programmable interface.
};
```

Clst41I and CLst41 class implementation

```
#include <objbase.h>
#include "resource.h"
#include "lst41c.h"
BSTR CreateBSTR(char *lpString)
{
    BSTR bsz;
    UINT cch;
    cch = strlen(lpString);
    bsz = SysAllocStringLen(NULL , cch);
    if(bsz == NULL)
        return NULL;
```

```
    if(cch > 0)
        MultiByteToWideChar(CP_ACP, 0, lpString, cch, bsz, cch);
    bsz[cch] = NULL;
    return bsz;
}
```

1. Function CreateBSTR will convert an ASCII string to a wide character (Unicode) string. In particular, function MultiByteToWideChar will convert an ASCII string pointed by lpString to a wide character string pointed by bsz. CP_ACP specifies the code page to be used by the conversion. It stands for ANSI code page. Other code page includes CP_MACCP that indicates a Macintosh code page, and CP_OEMCP that indicates an OEM code page.

```
STDMETHODIMP_(BSTR)
CLst41I::GetMachineName()
{
    BSTR b;
    ULONG ulLen;
    char *lpName;
    lpName = new char[MAX_PATH];
    ulLen = MAX_PATH;
    GetComputerName(lpName, &ulLen);
    b = CreateBSTR(lpName);
    return b;
}
//standard IUunknown methods implementation
STDMETHODIMP CLst41I::QueryInterface(REFIID riid, void** ppv)
{
    return m_pLst41->QueryInterface(riid, ppv);
}
STDMETHODIMP_(ULONG) CLst41I::AddRef()
{
    return m_pLst41->AddRef();
```

```
}
STDMETHODIMP_(ULONG) CLst41I::Release()
{
    return m_pLst41->Release();
}
IUnknown FAR*
CreateDispatchInterface(IUnknown* punkController,  void * pProgInterface)
{
    HRESULT hresult;
    ITypeLib* ptlib;
    ITypeInfo* ptinfo;
    IUnknown* punkStdDisp;
    hresult = LoadRegTypeLib(LIBID_Lst41, 1, 0, 0x0409, &ptlib);
    if (hresult != S_OK)
    {
        if((hresult = LoadTypeLib(L"lst41.tlb", &ptlib)) != S_OK)
            return NULL;
    }
    hresult = ptlib->GetTypeInfoOfGuid(IID_ILst41, &ptinfo);
    if (hresult != S_OK)
        return NULL;
    ptlib->Release();
    hresult = CreateStdDispatch(punkController,pProgInterface,
                                ptinfo, &punkStdDisp);
    if (hresult != S_OK)
        return NULL;
    ptinfo->Release();
    return punkStdDisp;
```

```
}
```

1. Function CreateDispatchInterface will first use registry information to load the type library by invoking the LoadRegTypeLib function. There are four parameters. LIBID_Lst41 is the Library ID being loaded. 1 is the type library (lst41.tlb)'s major version number. 0 is the lst41.tlb's minor version number. 0x0409 is the U.S English, which is the library's national language code. ptlib is a indirect pointer to ITypeLib interface.
2. If the type library information cannot be loaded from the registry, LoadTypeLib function will be called. LoadTypeLib loads and registers the type library stored in lst41.tlb.
3. GetTypeInfoOfGuid will retrieve IID_ILst41's type description, and return ptinfo, which is a indirect pointer to ITypeInfo.
4. After successfully invoking GetTypeInfoOfGuid, the CreateStdDispatch function will be invoked. This function creates a standard implementation of the IDispatch interface through one single function call. There are four parameters in this function. punkController is a pointer to the ILst41 IUnknown implementation. pProgInterface is a pointer to the object to expose. ptinfo is a pointer to the ILst41's type information that describes the exposed object. punkStdDisp is an indirect pointer to the ILst41 IDispatch interface implementation.

CLst41::CLst41()

```
{
    m_refs = 1;
    m_disp_interface = NULL;
    m_prog_interface = new CLst41I;
    m_prog_interface->m_pLst41 = this;
}
```

CLst41::~CLst41()

```
{
    delete m_prog_interface;
}
CLst41 * CLst41::Create()
{
    CLst41* pLst41;
    IUnknown* punkStdDisp;
    pLst41 = new CLst41();
    if(pLst41 == NULL)
      return NULL;
    punkStdDisp = CreateDispatchInterface((IUnknown *) pLst41,
                        pLst41->m_prog_interface);
```

```
    if (punkStdDisp == NULL) {
      pLst41->Release();
      return NULL;
    }
    pLst41->m_disp_interface = punkStdDisp;
    return pLst41;
}
STDMETHODIMP CLst41::QueryInterface(REFIID riid, void ** ppv)
{
    if (riid == IID_IUnknown)
        *ppv = this;
    else if (riid == IID_IDispatch || riid == DIID_DLst41)
        return m_disp_interface->QueryInterface(IID_IDispatch, ppv);
    else if (riid == IID_ILst41)
        *ppv = &m_prog_interface;
    else
    {
          *ppv = NULL;
        return ResultFromScode(E_NOINTERFACE);
    }
    AddRef();
    return S_OK;
}
```

1. QueryInterface will check the interface identifier riid. If riid is equal to IID_IUnknown, this value will be assigned to *ppv because CLst41 is the controlling IUnknown. If riid is equal to IID_IDispatch or DIID_DLst41 or IID_ILst41, *ppv will be the standard dispatch interface. Otherwise, *ppv will be NULL and the error code E_NOINTERFACE will be returned.

```
STDMETHODIMP_(ULONG) CLst41::AddRef()
{
```

```
    return ++m_refs;

}

STDMETHODIMP_(ULONG) CLst41::Release()

{

    if(--m_refs == 0)

    {

        if(m_disp_interface != NULL)

            m_disp_interface->Release();

        PostQuitMessage(0);

        delete this;

        return 0;

    }

    return m_refs;

}
```

1. PostQuitMessage function will indicate to the system that the thread will be terminated by posting a WM_QUIT message to the thread's message queue and return. When the thread receives the WM_QUIT from the message queue, it will terminate the message loop and return the control to the window.
2. 4. Implement the Class Object for CLst41.
3. Listing 4.4 demonstrates the CLst41 class factory implementation.

Listing 4.4. CLst41 class factory.

```
class CLst41CF : public IClassFactory

{

  public:

    CLst41CF();

    static IClassFactory* Create();

    STDMETHOD(QueryInterface)(REFIID riid, void ** ppv);

    STDMETHOD_(ULONG, AddRef)(void);

    STDMETHOD_(ULONG, Release)(void);

    STDMETHOD(CreateInstance)(    IUnknown* punkOuter,

                REFIID riid,
```

```
                    void** ppv);
    STDMETHOD(LockServer)(BOOL fLock);
  private:
    ULONG m_refs;
};
CLst41CF::CLst41CF()
{
    m_refs = 1;
}
IClassFactory* CLst41CF::Create()
{
    return new  CLst41CF();
}
STDMETHODIMP CLst41CF::QueryInterface(REFIID riid, void** ppv)
{
    if(riid == IID_IUnknown || riid == IID_IClassFactory)
    {
        AddRef();
        *ppv = this;
        return S_OK;
    }
    *ppv = NULL;
    return ResultFromScode(E_NOINTERFACE);
}
STDMETHODIMP_(ULONG) CLst41CF::AddRef()
{
    return ++m_refs;
}
```

```
STDMETHODIMP_(ULONG) CLst41CF::Release()
{
    if(--m_refs == 0)
    {
       delete this;
       return 0;
    }
    return m_refs;
}
STDMETHODIMP CLst41CF::CreateInstance(IUnknown* punkOuter,REFIID riid,
                                      void** ppv)
{
    extern CLst41 * g_pLst41;
    return g_pLst41->QueryInterface(riid, ppv);
}
STDMETHODIMP CLst41CF::LockServer(BOOL fLock)
{
    return S_OK;
}
```

1. 5. Create the main entry.
2. An Automation server can be one of two types. One is DLL based and can be driven only by automation clients. The other is .EXE based and can run standalone.
3. Lst41.exe is an .EXE-based automation server. To create a .EXE-based automation server, you must provide a main entry. Listing 4.5 demonstrates how to implement the main entry.

Listing 4.5. Main entry for Lst41.exe.

```
#include <objbase.h>
#include "lst41c.h"
#include <stdio.h>
CLst41 FAR* g_pLst41 = NULL;
```

```
void  main()
{
    MSG msg;
    DWORD g_dwLst41CF = 0;
    HRESULT hr;
    IClassFactory FAR* pcf;
    if((hr = OleInitialize(NULL)) != S_OK)
    {
        printf("OleInitialize Failed [0x%x]\n", hr);
        return;
    }
    if((g_pLst41 = CLst41::Create()) == NULL)
        return;
    pcf = CLst41CF::Create();
    if (pcf == NULL)
        goto Clean;
    hr = CoRegisterClassObject(CLSID_CLst41,
                               pcf,
                               CLSCTX_LOCAL_SERVER,
                               REGCLS_MULTIPLEUSE,
                               &g_dwLst41CF);
    if (hr != NOERROR)
        goto Clean;
    pcf->Release();
    while(GetMessage(&msg, NULL, NULL, NULL))
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
```

```
    }
Clean:
    if(g_dwLst41CF != 0)
        CoRevokeClassObject(g_dwLst41CF);
    if (g_pLst41 != NULL)
        g_pLst41->Release();
    OleUninitialize();
}
```

1. In Listing 4.5, first OleInitialize is invoked to initialize the OLE library. This function must be called before calling any OLE functions. Then CLst41::Create is invoked to create a single global instance of CLst41. CLst41CF::Create is called to create an instance of the class factory for CLst41. Register the class factory by invoking CoRegisterClassObject**.** Then the message loop is provided. When the WM_QUIT message is received, the message loop will be terminated. CoRevokeClassFactory will be called to inform OLE that the object is no longer available if the class factory was successfully created earlier. Finally, OleUninitialize will be called to uninitialize the OLE library and release all the resources.
2. 6. Create the registration entry.
3. lst41.exe has to be registered before being used. Listing 4.6 shows the registration file for lst41.exe.

Listing 4.6. lst41.reg file

```
REGEDIT
HKEY_CLASSES_ROOT\Lst41.Application.1 = Lst41 Automation Server
HKEY_CLASSES_ROOT\Lst41.Application.1\Clsid =
{9FBBEDE5-1B40-11d0-88E0-00AA004A7C7B}
HKEY_CLASSES_ROOT\CLSID\{9FBBEDE5-1B40-11d0-88E0-00AA004A7C7B} =
 IDispatch Lst41
HKEY_CLASSES_ROOT\CLSID\{9FBBEDE5-1B40-11d0-88E0-00AA004A7C7B}\ProgID =
Lst41.Application.1
HKEY_CLASSES_ROOT\CLSID\{9FBBEDE5-1B40-11d0-88E0-00AA004A7C7B}\
VersionIndependentProgID = Lst41.Application
HKEY_CLASSES_ROOT\CLSID\{9FBBEDE5-1B40-11d0-88E0-00AA004A7C7B}\
LocalServer32 =c:\ch4\lst41\debug\lst41.exe /Automation
; registration info Lst41 TypeLib
HKEY_CLASSES_ROOT\TypeLib\{9FBBEDE2-1B40-11d0-88E0-00AA004A7C7B}
```

```
HKEY_CLASSES_ROOT\TypeLib\{9FBBEDE2-1B40-11d0-88E0-00AA004A7C7B}\
1.0 = Lst41 Type Library
HKEY_CLASSES_ROOT\TypeLib\{9FBBEDE2-1B40-11d0-88E0-00AA004A7C7B}\
1.0\HELPDIR =
;Localized language is US english
HKEY_CLASSES_ROOT\TypeLib\{9FBBEDE2-1B40-11d0-88E0-00AA004A7C7B}\
1.0\409\win32 = c:\ch4\lst41\lst41.tlb
HKEY_CLASSES_ROOT\Interface\{9FBBEDE4-1B40-11d0-88E0-00AA004A7C7B} = DLst41
HKEY_CLASSES_ROOT\Interface\{9FBBEDE4-1B40-11d0-88E0-00AA004A7C7B}\
ProxyStubClsid = {00020420-0000-0000-C000-000000000046}
HKEY_CLASSES_ROOT\Interface\{9FBBEDE4-1B40-11d0-88E0-00AA004A7C7B}\
NumMethod = 7
HKEY_CLASSES_ROOT\Interface\{9FBBEDE4-1B40-11d0-88E0-00AA004A7C7B}\
BaseInterface = {00020400-0000-0000-C000-000000000046}
```

1. In Listing 4.6, Lst41.Application.1 is the ProgID, which is required for any automation objects. It is used by an automation controller to reference an automation server. LocalServer32 subkey specifies the full path to the 32-bit automation server lst41.exe.

Use Lst41 in C++ Application: lst41use.exe

lst41use.exe is a C++ application that uses lst41.exe. It uses IDispatch to access exposed objects.

Listing 4.7 demonstrates how to use IDispatch to access the methods exposed by lst41.exe.

Listing 4.7. lst41use.cpp

```
#include <objbase.h>
#include <initguid.h>
#include <stdio.h>
DEFINE_GUID(CLSID_CLst41,0x9FBBEDE5L,0x1B40,0x11D0,0x88,0xE0,
0x00,0xAA,0x00,0x4A,0x7C,0x7B);
LPSTR BstrToSz(LPCOLESTR pszW)
```

```
{
    ULONG cbAnsi, cCharacters;
    DWORD dwError;
    LPSTR lpString;
    if(pszW == NULL)
        return NULL;
    cCharacters = wcslen(pszW) + 1;
    cbAnsi = cCharacters * 2;
    lpString = (LPSTR) CoTaskMemAlloc(cbAnsi);
    if(NULL == lpString)
        return NULL;
    if(WideCharToMultiByte(CP_ACP, 0, pszW, cCharacters, lpString,
                           cbAnsi, NULL, NULL) == 0)
    {
        dwError = GetLastError();
        CoTaskMemFree(lpString);
        lpString = NULL;
    }
    return lpString;
}
```

Function BstrToSz converts a wide-character (Unicode) string to an ASCII string. CoTaskMemAlloc allocates a memory block using the default allocator. It behaves the same way as IMalloc::Alloc. The application should always check the return value from this function. Function WideCharToMultiByte maps a wide character string pointed by pszW to an ASCII string pointed by lpString.

```
void main()
{
    HRESULT hr;
    IDispatch *pIDispatch;
    DISPPARAMS dispparms = {NULL, NULL, 0,0};
```

```
    DISPID dispidGetMachineName;
    OLECHAR *pGetMachineName = L"GetMachineName";
    IUnknown *pIUnknown;
    VARIANT varResult;
    hr = OleInitialize(NULL);
    hr = CoCreateInstance(CLSID_CLst41, 0, CLSCTX_SERVER, IID_IUnknown,
                        (void**)&pIUnknown);
    if(FAILED(hr))
        printf("the error is %x\n", hr);
    pIUnknown->QueryInterface(IID_IDispatch, (void**)&pIDispatch);
    pIUnknown->Release();
    pIDispatch->GetIDsOfNames(IID_NULL,
                            &pGetMachineName,
                            1, LOCALE_SYSTEM_DEFAULT, &dispidGetMachineName);
    pIDispatch->Invoke(dispidGetMachineName, IID_NULL, LOCALE_SYSTEM_DEFAULT,
                       DISPATCH_METHOD, &dispparms,
                    &varResult, NULL, NULL);
    printf("the striing is %s\n", BstrToSz(varResult.bstrVal));
    pIDispatch->Release();
    CoUninitialize();
}
```

Function CoCreateInstance creates an CLst41 object. Because lst41.exe is a local server, execution context CLSCTX_SERVER is used. CLSCTX_SERVER is defined as

```
#define CLSCTX_SERVER (CLSCTX_INPROC_SERVER| CLSCTX_LOCAL_SERVER|
CLSCTX_REMOTE_SERVER)
```

GetIDsOfNames retrieves the DISPID and will be stored in dispidGetMachineName. There are five parameters in GetIDsOfNames. IID_NULL must be NULL; it is reserved for future use. pGetMachineName points to the method name. 1 indicates there is only one name to be mapped. LOCALE_SYSTEM_DEFAULT indicates the locale context in which to interpret the name.

Invoke accesses the GetMachineName method by providing its DISPID varResult is used to hold the return value from GetMachineName method.

Use lst41 in Visual Basic Application

With Visual Basic, using lst41.exe is straightforward. Listing 4.8 demonstrates how to use lst41.exe.

Listing 4.8. lst41.exe used in Visual Basic.

```
Dim x As Object

Dim strMachineName As String

Set x = CreateObject("lst41.application.1")

strMachineName = x.getmachinename

MsgBox strMachineName
```

In Listing 4.8, variable x is declared as an object and assigned the return of CreateObject call. The parameter in the CreateObject call is the ProgID of lst41.exe.

After the automation server (lst41.exe) is instantiated, the method GetMachineName can be invoked; the return value is assigned to strMachineName.

To try the example, place a CommandButton control on a form, and type Listing 4.8 into the command button's click proc. Run the example and click the Command1 button. A dialog box with the computer name on which the application is running will be displayed.

Create lst41 by Using the MFC ClassWizard (exe): lst42.exe

There are two MFC ClassWizard option provided by the New Project Workspace as shown in Figure 4.1.

Figure 4.1. New Project Workspace.

MFC AppWizard is designed to configure the skeleton of a new C++ application using the MFC.

MFC AppWizard (exe) is designed to create MFC extension .EXE, whereas MFC AppWizard (dll) is designed to create an MFC extension .DLL. The following steps illustrate how to create lst42.exe.

1. 1. Choose File|New. In the New dialog box, select the file type "Project Workspace."
2. 2. In the New Project Workspace dialog box, choose *the* MFC AppWizard (exe) in the type box. Type lst42 in the Name edit box and click the Create... button shown in Figure 4.2.

Figure 4.2. New Project Workspace dialog box.

1. 3. Choose the Single document option shown in Figure 4.3.

Note

The single-document option allows the application to work with one document at a time

---

The multiple-document option allows the application to work with multiple documents, each document with its own view.

Figure 4.3. MFC AppWizard—Step 1.

1. 4. Choose the Mini-server option and the OLE automation as shown in Figure 4.4.

Figure 4.4. MFC AppWizardmdStep 3 of 6.

Note

The Mini-server option allows the application to create and manage the compound document object. Mini-server cannot run standalone, and only supports embedded objects, whereas Full-server can run standalone and supports both linked and embedded objects.

The OLE automation option allows the application to be accessed by the automation clients, such as Visual Basic, Access, Excel.

1. 5. Click the Finish button. The files listed in Figure 4.5 will be generated by the MFC AppWizard (exe). Click the OK button to generate all the files.

Figure 4.5. New Project Information.

1. Among the files listed in Figure 4.5, the following classes and source files are specific to OLE:
2. SrvrItem.h, SrvrItem.cpp: This is the class that connects CLst42Doc to the OLE system. It also optionally provides links to the document.
3. IpFrame.h, lpFrame.cpp: This class is derived from COleIPFrameWnd and controls all frame features during in-place activation.
4. lst42.reg: This is a .REG file. This file can be used to manually register the application.
5. lst42.odl: This is the .ODL file, which is read by MkTypLib to create a type library (.TLB). A Type library is used by automation clients to retrieve information about the server's properties and its data types, methods, and its return value and parameters. Listing 4.9 shows the lst42.odl file.

Listing 4.9. Lst42.odl.

```
// lst42.odl : type library source for lst42.exe

// This file will be processed by the Make Type Library (mktyplib) tool to

// produce the type library (lst42.tlb).

[ uuid(06B70DA1-1818-11D0-A6AD-00AA00602553), version(1.0) ]

library Lst42

{

    importlib("stdole32.tlb");

    //  Primary dispatch interface for CLst42Doc

    [ uuid(06B70DA2-1818-11D0-A6AD-00AA00602553) ]

    dispinterface ILst42

    {
```

```
            properties:
                // NOTE - ClassWizard will maintain property information here.
                //    Use extreme caution when editing this section.
                //{{AFX_ODL_PROP(CLst42Doc)
                //}}AFX_ODL_PROP
            methods:
                // NOTE - ClassWizard will maintain method information here.
                //    Use extreme caution when editing this section.
                //{{AFX_ODL_METHOD(CLst42Doc)
                [id(1)] BSTR GetMachineName();
                //}}AFX_ODL_METHOD
        };
        //  Class information for CLst42Doc
        [ uuid(06B70DA0-1818-11D0-A6AD-00AA00602553) ]
        coclass Document
        {
            [default] dispinterface ILst42;
        };
        //{{AFX_APPEND_ODL}}
    };
```

1. ODL consists of attributes, statements, and directives.

```
    [ uuid(06B70DA1-1818-11D0-A6AD-00AA00602553), version(1.0) ]
```

1. is the attribute, which associates information with the library. The uuid is for the type library. All applications that expose type information must register the information to the system registry so that it is available to type browsers or any automation clients.
2. The definition on the library Lst42 is enclosed between { and }. In the definition. import indicates that lst42.tlb imports the standard OLE library stdole32.tlb. dispinterface defines a set of methods and properties that can be invoked by IDispatch::Invoke. coclass named Document indicates that supported interface ILst42 in this component object (lst42.exe). A GUID must be given on a coclass. This GUID is the same as CLSID registered in the system.
3. 6. From the View menu, choose the ClassWizard command, select the OLE Automation tab, and choose CLst42Doc in the Class name dropdown list box shown in Figure 4.6.

Figure 4.6. MFC ClassWizard—OLE Automation.

1. 7. Click the Add Method... button in Figure 4.6. Type GetMachineName in the External name dropdown combo box, select BSTR in the Return type dropdown list box, and click OK button in the Add Method dialog box displayed in Figure 4.7.

Figure 4.7. Add Method dialog box.

Note

External name is used by the automation clients to invoke the exposed method, whereas Internal name is the member function that implements the exposed method.

---

1. 8. Modify the class lst42Doc.cpp. Add the code shown in Listing 4.10.

Listing 4.10. Method implementation.

```
BSTR CLst42Doc::GetMachineName()
{
    CString strResult;
    ULONG ulLen;
    char *lpName;
    lpName = new char[MAX_PATH];
    ulLen = MAX_PATH;
    GetComputerName(lpName, &ulLen );
    strResult = lpName;
    delete [] lpName;
    return strResult.AllocSysString();
}
```

1. Lst42.exe only has one exposed method, GetMachineName. Listing 4.11 demonstrates the dispatch map generated by the MFC AppWizard (exe).

Listing 4.11. Dispatch map for lst42.exe.

```
BEGIN_DISPATCH_MAP(CLst42Doc, COleServerDoc)
    //{{AFX_DISPATCH_MAP(CLst42Doc)
    DISP_FUNCTION(CLst42Doc, "GetMachineName", GetMachineName,
     VT_BSTR, VTS_NONE)
    //}}AFX_DISPATCH_MAP
```

```
END_DISPATCH_MAP()
```

1. Listing 4.11 is abridged from lst42Doc.cpp to emphasize the essential parts. DISP_FUNCTION macro is used in the dispatch map to define an exposed method. The dispatch map is a mechanism provided by MFC to dispatch the request made by the automation clients, such as calling the methods and accessing the properties.
2. The Dispatch map indicates the external and internal name of the properties and methods, as well as the properties' data types and method's argument and return type.
3. In Listing 4.11, CLst42Doc is the name of the class. COleServerDoc is the base class. DISP_FUNCTION macro is used to define an automation method. GetMachineName is the external name used by the automation clients, GetMachineName is the internal name, VT_BSTR is the return type, VTS_NONE is the method's parameter list. In this case, there is no input parameter.
4. Besides the DISP_FUNCTION macro defined in the dispatch map, other macros are provided as shown in the following list:
5. DISP_PROPERTY defines an automation property.
6. DISP_PROPERTY_EX defines an automation property and names the "get" and "set" functions.
7. DISP_PROPERTY_NOTIFY defines an automation property with notification.
8. DISP_PROPERTY_PARAM defines an automation property, names the "get" and "set" functions, and an index parameter.
9. DECLARE_DISPATCH_MAP is used in the class declaration to indicate that a dispatch map will be used.
10. For example,

```
class CLst42Doc : public COleServerDoc


...
    // Generated OLE dispatch map functions
    //{{AFX_DISPATCH(CLst42Doc)
    afx_msg BSTR GetMachineName();
    //}}AFX_DISPATCH
    DECLARE_DISPATCH_MAP()
    DECLARE_INTERFACE_MAP()
}
```

1. is used in the lst42Doc.h to indicate that a dispatch map will be used in CLst42Doc class.

- BEGIN_DISPATCH_MAP is used in the class implementation to indicate the start of the dispatch map definition as shown in Listing 4.11.
- END_DISPATCH_MAP is used in the class implementation to indicate the end of the dispatch map definition as shown in Listing 4.11.

1. 9. Build the project to generate lst42.exe.
2. Before lst42.exe can be used by the automation clients or any other information, lst42.exe must be registered with the system. This can be done by running one of the following commands:
3. regedit /s lst42.reg
4. lst42.exe /regserver

Note

> Automation server generated by the MFC AppWizard (exe) provides the self registration features. In other words, lst42.exe accepts the program argument /regserver to register itself to the system registry.

After lst42.exe is registered with the system, it can be used by automation clients, such as Visual Basic, Access, Excel. lst42.exe can be accessed by using IDispatch interface in the C++ application.

The following example demonstrates how lst42.exe be used in Visual Basic 4.0.

With Visual Basic, using lst42.exe is straightforward. Listing 4.12 demonstrates how to use lst42.exe inside Visual Basic.

Listing 4.12. lst42.exe used in Visual Basic.

```
Dim x As Object

Dim strMachineName As String

Set x = CreateObject("lst42.document")

strMachineName = x.getmachinename

MsgBox strMachineName
```

In Listing 4.12, variable x is declared as an object and assigned the return of the CreateObject call. The parameter in the CreateObject call is the ProgID of lst42.exe. For any application generated by the MFC AppWizard with OLE automation enabled, the ProgID is always Name for the new project workspace plus the .document.

After the automation server (lst42.exe) is instantiated, the method GetMachineName can be invoked and the return value is assigned to strMachineName.

To try the example, place a CommandButton control on a form and type Listing 4.13 into the command button's click proc. Run the example and click the Command1 button. A dialog box with the name of the computer the application is running on will be displayed.

Create lst41 by Using the Control Wizard: lst43.ocx

Besides using MFC AppWizard (exe) to create the automation server, Control Wizard can be used. The IsInvokeAllowed() method is required to be overriden to support the automation.

The following steps demonstrates how to implement an automation server supporting the same functionality as lst41.exe.

1. 1. Choose File|New. In the New dialog box, select the file type "Project Workspace."
2. 2. In the New Project Workspace dialog box, choose the OLE ControlWizard. Type lst43 in the Name edit box and click the Create… button shown in Figure 4.8.

Figure 4.8 New Project Workspace—OLE Control Wizard

1. 3. Click the Finish button. Files shown in Figure 4.9 will be generated. Click the OK button in the dialog box displayed in Figure 4.9.

Figure 4.9. Files generated by OLE ControlWizard.

Among the files shown in Figure 4.9, the following files are specifically related to the Control.

> lst43.odl: containing .ODL script.
>
> lst43Ctl.h, lst43Ctl.cpp: all the methods, properties, and events will be placed in this class.

lst43Ppg.h, lst43Ppg.cpp: default property page class.

lst43Ctl.bmp: containing a bitmap displayed in the container's toolbox, such as Visual Basic's toolbox.

1. 4. Choose View|ClassWizard. Select the OLE Automation tab shown in Figure 4.10, click the Add Method… button, enter GetMachineName, and choose BSTR as the return type in the Add Method dialog box displayed in Figure 4.11.

Figure 4.10. OLE Automation tab in MFC ClassWizard.

Figure 4.11. Add Method in OLE Automation tab.

1. 5. Implement the GetMachineName method by modifying lst43Ctl.cpp.

```
BSTR CLst43Ctrl::GetMachineName()
{
    CString strResult;
    ULONG ulLen;
    char *lpName;
    lpName = new char[MAX_PATH];
    ulLen = MAX_PATH;
    GetComputerName(lpName, &ulLen );
    strResult = lpName;
    delete [] lpName;
    return strResult.AllocSysString();
}
```

1. 6. Override IsInvokeAllowed.
2. Add the following declaration in class CLst43Ctrl in lst43Ctl.h.

```
private:
    BOOL IsInvokeAllowed(DISPID dispid);
```

1. Add the IsInvokeAllowed implementation in lst43Ctl.cpp.

```
BOOL CLst43Ctrl::IsInvokeAllowed(DISPID dispid)
{
    return TRUE;
}
```

1. 7. Build the application to generate lst43.ocx.
2. lst43.ocx can be used in the Visual Basic the same way as lst42.exe. Instead of Listing 4.12, code in Listing 4.13 should be

entered.

Listing 4.13. lst43.ocx used in Visual Basic.

```
Dim x As Object

Dim strMachineName As String

Set x = CreateObject("lst43.lst43ctrl.1")

strMachineName = x.getmachinename

MsgBox strMachineName
```

1. Lst43.ocx supports self registration. To register lst43.ocx, run regsvr32 lst43.ocx, to unregister lst43.ocx, run regsvr32 /u lst43.ocx.
2. In Listing 4.13, ProgID for lst43.ocx is lst43.lst43ctrl.1. The default ProgID generated by OLE ControlWizard is always the name for the new project workspace plus .lst43ctrl.1.
3. There is one method exposed by lst43.ocx. Listing 4.14 is the dispatch map generated by the ClassWizard.

Listing 4.14. Dispatch map for lst43.ocx.

```
BEGIN_DISPATCH_MAP(CLst43Ctrl, COleControl)

    //{{AFX_DISPATCH_MAP(CLst43Ctrl)

    DISP_FUNCTION(CLst43Ctrl, "GetMachineName", GetMachineName,

    VT_BSTR, VTS_NONE)

    //}}AFX_DISPATCH_MAP

    DISP_FUNCTION_ID(CLst43Ctrl, "AboutBox", DISPID_ABOUTBOX, AboutBox,

    VT_EMPTY, VTS_NONE)

END_DISPATCH_MAP()
```

1. In Listing 4.14, DISP_FUNCTION is exactly the same as in Listing 4.12, except the class name is CLst43Ctrl instead of CLst42Doc.

Create lst41 by Using the Active Template Library: lst44.dll

ATL provides an OLE COM AppWizard to create COM objects. It supports COM objects with a custom interface, IDispatch, and IConnectionPoint and so on. ATL is designed to create COM objects. For more information on ATL, please refer to Appendix E, "ActiveX Template Library."

The following example illustrates how to use ATL to create an automation server supporting the same functionality as lst41.exe.

1. 1. Choose File|New. In the New dialog box, select "Project Workspace" in the type box.
2. 2. In the New Project Workspace dialog box, choose the OLE COM AppWizard. Type lst44 in the Name edit box and click the Create… button shown in Figure 4.12.

Figure 4.12. OLE COM AppWizard.

1. 3. Use the default option displayed in Figure 4.13.

Figure 4.13. OLE COM AppWizard—Step 1 of 2.

Note

The Dual Interface option indicates that the interface supports IDispatch and IUnknown.

---

1. The Custom Interface option indicates the interface only supports IUnknown.
1. 4. Implement the GetMachineName method
2. Add the following code in bold font to interface ILst44 definition in lst44.idl.

```
interface ILst44 : IDispatch
{
    import "oaidl.idl";
    HRESULT GetMachineName([out,retval] BSTR *retval);
}
```

1. Add the following code in bold font to interface ILst44 definition in lst44.odl.

```
interface ILst44 : IDispatch
{
    HRESULT GetMachineName([out,retval] BSTR *retval);
};
```

1. Add the following code in CLst44Object.h in lst44obj.h.

```
public:
    STDMETHOD(GetMachineName)(BSTR *retval);
```

1. Add GetMachineName implementation in lst44obj.cpp shown in Listing 4.15.

Listing 4.15. Addition to lst44obj.cpp.

```
BSTR CreateBSTR(LPCSTR lpa)
{
    BSTR bsz;
    UINT cch;
    cch = strlen(lpa);
```

```
        bsz = SysAllocStringLen(NULL, cch);
        if (bsz == NULL)
            return NULL;
        if (cch > 0)
            MultiByteToWideChar(CP_ACP, 0, lpa, cch, bsz, cch);
        bsz[cch] = NULL;
        return bsz;
    }
    STDMETHODIMP CLst44Object::GetMachineName(BSTR* retval)
    {
        ULONG ulLen;
        char *lpName;
        lpName = new char[MAX_PATH];
        ulLen = MAX_PATH;
        GetComputerName(lpName, &ulLen);
        if(ulLen == 0)
            *retval = NULL;
        else
            *retval = CreateBSTR(lpName);
        return S_OK;
    }
```

1. In Listing 4.15, function CreateBSTR accepts an ASCII string, and converts into a Unicode string. Unicode stands for a 16-bit character set that can encode all known character sets and is used as a world-wide character encoding standard.
2. 5. Before building the project, using the midl lst44.idl command to generate a source file for a custom OLE interface.
3. 6. Build the project to generate lst44.dll.

lst44.dll can be used in Visual Basic the same way as lst42.exe. Instead of input Listing 4.11, code in Listing 4.16 should be entered.

Listing 4.16. Lst44.dll used in Visual Basic.

```
Dim x As Object
Dim strMachineName As String
```

```
Set x = CreateObject("lst44.lst44object.1")

strMachineName = x.getmachinename

MsgBox strMachineName
```

Before running the preceding code, lst44.dll needs to be registered by running regsvr32 lst44.dll.

In Listing 4.16, ProgID for lst44.dll is lst44.lst44object.1. The default ProgID generated by OLE COM AppWizard is the name of the project workspace plus .lst44object.1.

The default ProgID can be modified by replacing the code in bold font as shown in the following; this code is contained in lst44.cpp.

```
BEGIN_OBJECT_MAP(ObjectMap)

    OBJECT_ENTRY(CLSID_Lst44, CLst44Object, "LST44.Lst44Object.1",

    "LST44.Lst44Object.1", IDS_LST44_DESC, THREADFLAGS_BOTH)

END_OBJECT_MAP()
```

Use lst44.dll on the Internet Explorer 3.0 and Web Server

Reusable components such as automation server not only can be used in the automation clients such as Visual Basic, Access, Excel, or in the application that accesses the automation server via IDispatch. Microsoft Internet Explorer (IE) 3.0 also supports the use of automation server.

The automation server inside IE 3.0 requires an <OBJECT> tag to be used to include the object.

Listing 4.17 demonstrates how to use lst44.dll inside an HTML page and displayed in IE 3.0 browser.

Listing 4.17. lst44.dll used in IE 3.0.

```
<HTML>

<HEAD>

<OBJECT  classid="clsid:C566CC25-182E-11D0-A6AD-00AA00602553"

    id= MachineName

</OBJECT>

<SCRIPT language="VBScript">

    msgbox MachineName.getmachinename

</SCRIPT>

</HEAD>
```

```
</HTML>
```

In Listing 4.17, "clsid:..." is the string representation of the CLSID, denoted as {CLSID}, for lst44.dll. The following steps illustrates how to get the {CLSID} for lst44.dll.

1. 1. Run regedt32
2. 2. Go to HKEY_CLASSES_ROOT and then find lst44.lst44object.1 subkey in HKEY_CLASSES_ROOT.
3. 3. Get the value of the CLSID subkey under lst44.lst44object.1, shown in Figure 4.14.

Figure 4.14. ProgID and {CLSID} registry key for lst44.dll.

1. 4. Double-click the data, and a string editor dialog box will be displayed shown in figure 4.15.

Figure 4.15. String Editor for data.

1. 5. Paste the value in string editor to the HTML page.
2. When IE 3.0 browses this page, a message box will pop up, showing the computer name.
3. Automation server can be used not only on the browser's side, but also on the Web server side.
4. To use the automation server on the Web server side, Internet Personalization System (IPS) needs to be installed on top of the Internet Information Server (IIS). IPS provides the environment to support the usage of automation servers. It also provides the intrinsic controls listed as follows:
5. The Request component is composed of three items within a collection: ServerVariables, QueryString, and Body. The collection can be accessed via

```
<% Request.[ServerVariables|QueryString|Body]("variable name") %>
```

1. The collection name is optional; if it is not provided, the server will search the collection in the following order:

```
ServerVariable, QueryString, Body.
```

1. The ServerVariables collection supports all HTTP headers by prefixing them with HTTP_ and the variables including AUTH_TYPE, CONTENT_LENGTH, CONTENT_TYPE, GATEWAY_INTERFACE, PATH_INFO, PATH_TRANSLATED, QUERY_STRING, REMOTE_ADDR, REMOTE_HOST, REMOTE_IDENT, REMOTE_USER, REQUEST_METHOD, SCRIPT_NAME, SERVER_NAME, SERVER_PORT, SERVER_PROTOCOL, and SERVER_SOFTWARE.
2. The HTTP headers can be found at http://www.w3.org.
3. The QueryString collection provides access to all parameters in the Get method.
4. The Body collection provides access to all parameters in the Post method
5. The Response components exposes methods or properties including Add(*header-value, header-name*), AppendToLog(*string*), Clear, Expires, Redirect(*url*), SetCookie(*name, value[expires, [domain,[path,[secure,]]]]*), and Status.
6. The Server components exposes three methods: HTMLEncode (string), Include(filename), and MapPath(vitual path).
7. Listing 4.18 demonstrates how to use lst44.dll on the Web server side so that the user can get the Web server machine name.

Listing 4.18. Use lst44.dll on Web server.
getmachinename.asp file

```
<% x = server.createobject("lst44.lst44object.1") %>

<% = x.getmachinename %>
```

machinename.html file

```
<HTML>

<BODY>

<A HREF="/scripts/machinename.asp"> Get the Server Machine Name</A>

</BODY>

</HTML>
```

1. In Listing 4.18, .asp stands for active server page. It is designated script files. The extension .asp will cause the Web server to invoke the IPS script interpreter. <% ... %> indicates that scripting language expressions, <% = %> indicates that the value of the expression will put into the HTML stream.
2. In Listing 4.18, getmachinename.asp should be placed under the scripts directory, and machinename.html placed under the wwwroot directory. When machinename.html page is launched and link "Get the Server Machine Name" is clicked, the Web server machine name will be displayed in the browser.

Summary

An OLE automation server is a COM server with the support of the IDispatch interface. Applications can use IDispatch to access exposed objects in automation server. A lot of tools can be used to develop the automation servers by using MFC AppWizard, Control Wizard, and Active Template Library. The automation server is a reusable component, which can be used in the automation controller, IE 3.0, and Web server.

Chapter 5

OLE Controls

by Vincent W. Mayfield

This chapter covers OLE Controls. OLE Controls, also called OLE Control Extensions, are commonly referred to as OCXs. OLE Controls are known as OCXs, for their file extension, and also as ActiveX Controls, which are OLE Controls extended for use in Internet applications. A developer can create ActiveX Controls for use in Internet applications, and those controls can also be utilized in non-Internet applications. ActiveX Controls are a superset of OLE Controls. Therefore, throughout this chapter, the terms OLE Control, OCX, and ActiveX Control are used somewhat interchangeably (see Figure 5.1).

Figure 5.1. OLE Controls, OCXs, and ActiveX Controls are terms that refer to similar entities.

If a control is an OLE Control, it is not necessarily an ActiveX Control. Conversely, though, if a control is an ActiveX Control it *is* an OLE Control. There are some distinct things that make a control an ActiveX Control. Not to over trivialize things, for all intents and purposes, OLE Controls and ActiveX Controls are the same except for a few differences. I don't want you to feel that ActiveX Controls are some entirely new thing as the marketing types would have us believe. This chapter highlights those differences. To start out with, take a look at the origins of the OLE Control.

A Short History

The term *control* or *custom control* has been around since Windows 3.0, when it was first defined. In fact, a custom control was nothing more than a Dynamic Link Library that exported a defined set of functions. Unlike a .DLL a custom control can manipulate properties and handle the firing of events in response to user or programmatic input.

The Visual Basic development environment had caught on in the development community. Custom controls were necessary because developers found they needed better ways to express the user interface of their applications, and many times, there was simply no way to perform a complex operation in Visual Basic. Unfortunately, or fortunately depending on your perspective, these C DLLs had no way of allowing Visual Basic to query the control for information on the properties and methods supported by the control. This made custom controls difficult to use in the Visual Basic development environment.

In 1991, Microsoft unveiled the VBX. The VBX stood for Visual Basic Extension. The idea was that these little reusable software components could be embedded in their container. To everyone's surprise, VBXs took off like wildfire. Companies cropped up all over the place developing these little reusable software components. VBXs were able to provide a wide range of functionality, from a simple text label to a complex multimedia or communications control. VBXs were written in C and C++ and provided a wide variety of capabilities that could not have been done in a Visual Basic application otherwise. VBXs became extremely popular.

Because VBXs had become popular, demand for them grew within the developer market. Soon developers wanted them for 32-bit applications and even on non-Intel platforms such as the DEC Alpha, RISC, Power PC, and the MIPS. Developers wanted to extend VBXs by using Visual Basic for Applications to connect VBXs with applications such as Access, PowerPoint, Excel, Project, and Word.

Unfortunately, VBXs are severely restricted. They are built on a 16-bit architecture that is not designed as an open interface. They were primarily designed to accommodate the Visual Basic environment. This made VBXs almost impossible to port to a 32-bit environment.

In 1993 OLE 2.0 was released. With the release of OLE 2.0, Microsoft extended the OLE architecture to include OLE Controls. OLE Controls, unlike their predecessors, the VBX and the custom control, are founded on a binary standard,

the Component Object Model. In addition, OLE Controls support both a 16- and 32-bit architecture.

Note

Kraig Brockschmidt wrote what is sometimes considered the bible for OLE programmers. The book is *Inside OLE*, published by Microsoft Press. The original title of the book was *Inside OLE 2.0*, but as you discovered in Chapter 2, "OLE Components," OLE is an ostensibly virtual standard building on each layer. Therefore, in the second edition, the 2.0 was dropped. *Inside OLE* thoroughly explores the OLE standard from the API level. Every OLE programmer should read *Inside OLE*.

Instead of creating an extended architecture for VBXs, Microsoft decided to develop the OCX to offer the benefits of a component architecture to a wider variety of development environments and development tools (See Figure 5.2). The Component Object Model and OLE are open architectures, giving them a wider variety of input from the industry. Like their predecessor the VBX, OLE Controls are also known by their file extension; OCXs (OLE Control Extension), likewise, have taken the market by storm.

Figure 5.2. The progression of development of the OLE and ActiveX Controls.

From 1993 to 1995, OLE Controls have flourished. Many Independent Software Vendors (ISVs) converted their VBXs to OLE Controls, and in some cases they maintained three versions; VBX, 16-bit OCX, and a 32-bit OCX. The makers of Visual C++ and MFC created the OLE Control Developers Kit, and even incorporated it into Visual C++ 2.0 and 1.5, further adding to the success of OLE Custom Control.

Between 1995 and 1996, the Internet took the world like a blitzkrieg, causing Internet Mania. Everyone had to become Web-enabled. Companies found themselves making Web sites because they saw the Internet as the great advertisement media for the year 2000 and beyond. Unfortunately, in previous years the Internet had been a relatively static environment. This is due in part to the Internet's roots with the big-iron diehards who grew up with the IBM Mainframes, the VAXs, and the UNIX boxes. However, PC computers have become household devices for the common person. Users have become accustomed to graphical interaction with their machines thanks to the Macintosh, Microsoft Windows, and X-Windows/Motif. Thanks to SUN and their invention of the Java programming language and the Java applet, the Internet is no longer a static environment. . Web pages exploded to life with multimedia, sound, and dynamic interaction.

Microsoft, realizing the potential and the hype surrounding the Internet Explosion, decided they needed to get with the program and take a role of leadership in this emerging environment. Microsoft boldly announced they were going to "Activate" the Internet in 1996 with ActiveX technologies (a little late, but better late than never). Thus, from these ActiveX technologies, the ActiveX Controls were born. ActiveX Controls were nothing really new, just an extension of their mother, the OLE Control. ActiveX Controls are simply OLE Controls implemented smarter and enhanced to be utilized across the Internet.

What Is an OLE Custom Control?

Now that you know a little of the history behind an OLE Control, this section explores just what an OLE Control is. An OLE Control is an embeddable Component Object Model object that is implemented as an in-process server dynamic link library. It supports in-place activation as an inside-out object.

Note

The title of the book *OLE Controls—Inside Out*, by Adam Denning, Microsoft Press is a play on words because OLE Controls are activated from the inside out. This book is also an

excellent reference.

As an OLE In-Proc Object, an OLE control is loaded into the address space of its container. As you are probably aware, every WIN32 process has a 4-gigabyte address space.

Note

A WIN32 Process is an running instance of an application loaded into memory.

The lower 2 gigabytes is where the application is loaded and the upper 2 gigabytes is where the system is loaded. An OLE Control is loaded in the lower 2 gigabytes with the application. Therefore, they share the same resources with the application; hence the term in-process.

An OLE Control is also a server. Why is it a server? Well, it provides two-way communication between the "container application" and the control. It can also respond to user-initiated events such as mouse movements, keyboard input, and programmatic scripting input, and it can pass that input to the container application for action.

OLE Controls are also in-place activated. This means that they can be placed in the active state by the user or the container and edited or manipulated. This is a functionality OLE Controls inherit from OLE Documents. Like a dynamic link library (DLL), the OLE Control is a library of functions. In fact, an OLE Control might be considered a "super DLL." More than just a "super DLL," an OLE Control is a detached object that can fire and respond to events, process messages, has unique properties, and possesses multi-threaded capabilities. OLE Controls are also known as OCXs because of their file extension, but they are actually a DLL. OCXs can contain several controls. Unlike DLLs, OCXs respond to user input and support two-way communication or notification, between themselves and their container.

An OLE Control can have its own data set and can act as an OLE Automation component because you can manipulate its properties and methods. OLE Controls can be both 16- and 32-bit as well as Unicode. OLE Controls, like OLE Automation objects, can have properties set at both compile time and runtime, and OLE Controls also have methods that can perform certain operations. The difference between OLE Controls and OLE Automation objects is that they are self-contained objects. They provide two-way communication between the control and the container. In addition, OLE controls do not have to have a user interface. As such they can provide hidden services such as a timer, communications, or mail.

OLE Controls cannot stand alone; they must be embedded in an OLE Container. OLE Controls provide prepackaged components of functionality that are reusable and customizable. OLE Controls are at the top of the OLE Architecture. Thus they are built on several OLE technologies. In addition, OLE Controls can be used in a wide variety of development tools, such as Delphi, Visual C++, Borland C++, Gupta, Visual Basic, Oracle Developer 2000, and PowerBuilder. OLE Controls can also be used in a variety of non-programming environments, such as Microsoft Word, Microsoft Excel, Lotus, HTML, and Internet Explorer. OLE Controls are a very powerful reusable component.

OLE Control Architecture

The beauty of OLE Controls is that they are programmable and reusable. They expose themselves to the outside world and can be utilized in a variety of programming and non-programming environments. An OLE Control is like an OLE Compound Document, but it is extended by using OLE Automation through IDispatch to support properties and methods. What makes OLE Controls unique is events. OLE Controls have three sets of attributes that are exposed to the outside world:

- Properties
- Methods

- Events

Properties

Properties are named attributes or characteristics of an OLE Control. These properties can be set or queried. Some examples of properties are color, font, number, and so on.

Usually OLE Controls provide access to their properties through property sheets. Property sheets are separate OLE Automation entities. This feature is not limited to design/compile time, but also can be displayed at runtime to allow the user to manipulate the control's properties, events, or methods. Property sheets are a user interface component that is a tabbed dialog. OLE Automation provides the mechanism by which controls communicate with their property sheets.

OLE Controls have what are called stock properties. These are properties common to all OLE Controls. If you are using the Base Control Framework provided in the ActiveX SDK or the ActiveX Template Library, you will have to implement the stock properties and their pages yourself. However, if you are using Microsoft Foundation Classes, you can take advantage of these stock properties because they are already built in. These are general (see Figure 5.3), color (see Figure 5.4), font (see Figure 5.5), and picture properties (see Figure 5.6).

Figure 5.3. A property sheet with the General stock properties.

Figure 5.4. A property sheet with the Font stock properties.

Figure 5.5. A property sheet with the Color stock properties.

Figure 5.6. A property sheet with the various Picture stock properties.

OLE Controls also have persistent properties. These properties are stored in the container and set at design or compile time. Controls also have the ability to save persistent information about their properties at runtime, and thus in effect can save their state. This means that the controls can also load their persistent properties at initial load time.

Events

Events are a notification triggered by the control in response to some external action on the control. Usually this is some input by the user such as a mouse click or keyboard input. That event is then communicated to the control's container by the controls. This is done through a communications mechanism known as Lightweight Remote Procedure Call (LRPC). LRPCs are the scaled-down brother of the Remote Procedure Call (RPC).

RPCs are an interprocess communications mechanism used to provide two-way communications between applications. This can be on the same computer or between computers across a network. RPC is the mechanism that Network OLE, also known as Distributed COM (DCOM), uses to exchange objects across process and computer boundaries. RPC is much more than just a communications method. It allows a function in a process on one computer to evoke a function in a process on another computer. This can even be computers across an enterprise-wide network or the Internet.

Lightweight Remote Procedure Calls, unlike their big brother RPCs, are only for communications between processes, or within processes on a single computer. LRPCs are the mechanism by which an OLE Control dispatches, through the IDispatch interface, control notifications to the container, and the reverse, from the container to the control. This communication is based on posting messages or events to window handles to transfer data between processes. It is also known as marshaling.

Methods

Methods are functions performed by the control to access the control's functionality. This allows some external source the ability to manipulate the appearance, behavior, or properties of the control. These are actions such as GetColor, SetColor, CutToClipBoard, PasteFromClipboard, and so on. Methods are inherited from OLE Automation. A method is the interface in which an application or a programmer can set or receive values from an OLE Control.

Methods are a lot like member functions in C++. They provide accessor functions that grant access to an OLE Control's properties and data. An OLE Control's properties are like a C++ class's member variables. Methods are both stock and custom, as are properties. Stock methods provide access to stock properties, such as color, font, and picture. Likewise, custom methods provide access to custom properties. With methods, you can change a control's appearance or initialize it with a value. Using Visual Basic or Visual C++, you can program a link between it and another application or control.

OLE Control Interfaces

Like all other COM objects, OLE Controls are manipulated through interfaces. In the original OLE Control and OLE Container specification, OLE Controls were required to support certain interfaces, whether they needed or utilized them or not. This left some controls bloated with code and overhead that they did not need.

Presently, the only interface a control is required to implement is the IUnknown. This is mentioned so that you realize that a new standard has been published. In December of 1995, Microsoft published the OLE Controls and OLE Container Guidelines Version 2.0. This was an extension of Version 1.1. With the advent of ActiveX Controls, the standard was changed to the 1996 standard for ActiveX Controls and ActiveX Containers and is again an extension to the previous standard. The next section discusses the specifics of an ActiveX Control.

An OLE Control exposes interfaces. Likewise, a container exposes interfaces to the OLE Control. OLE Controls and OLE Containers link through interfaces (see Figure 5.7).

Figure 5.7. OLE Controls and their Containers communicate through interfaces using LRPCs.

There are approximately 26 interfaces for OLE Controls and their Containers. The next section on ActiveX Controls discusses the new interfaces. This is not considered an all-inclusive list, as there are a few other interfaces that are used, but these represent the main interfaces.

In Table 5.1, notice that each object supports the IUnknown. This is now the only interface required to be supported by an OLE Control. However, if you implemented only the IUnknown, you would have a control that did pretty much nothing. The idea is to implement only the interfaces needed to support the control. In Figure 5.8 you can see how an OLE Control's interfaces relate to the container interfaces. In addition, when you write the code for your control, you must be cognizant of the interfaces the control supports, as you must also be cognizant that all OLE Containers do not support all interfaces. In order to be compatible with as many containers as possible, you must check for the support of your interfaces by the container and degrade your control's functionality gracefully in the event an interface is not supported. This can be likened to error checking, except that you still want your control to function, but with degraded capability or through an alternative interface.

Figure 5.8. How the OLE Control interfaces relate to the OLE Container interfaces.

The most important interfaces are IOleControl and IDispatch. IDispatch is the mechanism through which OLE Controls communicate. IOleControl encapsulates the basic functionality of an OLE Control. Table 5.1 shows the COM interfaces an OLE Control or an OLE Container may support in order to facilitate the operations between them.

Table 5.1. COM interfaces for facilitating operations between controls and containers.

OLE Control Control Site Client Site Container IClassFactory2 IOleControlSite IOleClientSite IOleInPlaceUIWindow IOleObject IUnknown IOleInPlaceSite IOleInPlaceFrame IDataObject IAdviseSink IUnknown IViewObject IDispatch IPersistStorage IUnknown IOleInPlaceActiveObject IOleCache IPersistStreamInit IOleControl IConnectionPointContainer IConnectionPoint IProvideClassInfo IProperNotifySink ISpecifyPropertyPages IPerPropertyBrowsing ISimpleFrameSite IDispatch IUnknown

The important thing to remember is that the interfaces a control supports define that control. However, you should implement only the interfaces your control requires to function. This idea will become more apparent in the following section on ActiveX Controls.

ActiveX Controls

An ActiveX Control is a superset of an OLE Control that has been extended for the Internet environment. This does not mean that ActiveX Controls can be utilized only in the Internet environment; quite the contrary, they can be utilized in any container that can support their interfaces. ActiveX Controls must still be embedded in a container application. When an end user encounters a page with an ActiveX Control, that control is downloaded to the client machine if it is not already there and used. This is, of course provided that the user's browser supports ActiveX controls. The two most prevalent browsers that support ActiveX Controls are Microsoft Internet Explorer and Netscape, with the help of the Ncompass plug-in.

The major difference between the OLE Control and the "superset" ActiveX Control is that the standard is different. In the new standard an ActiveX Control must support at least the IUnknown interface and be self-registering. It is a simple COM object. Obviously the control must have more interfaces than just IUnknown, or it would have no functionality. The idea is that the control support only the interfaces it needs, so it can be as lightweight as possible. In contrast, in the previous standard an OLE Control was required to support a whole armada of COM interfaces, whether the control needed them or not. This made some controls bloated with code that was not utilized or needed. In the world of Internet development this code bloat is unacceptable.

Supporting the IUnknown

The minimum interface for an ActiveX Control to support is the IUnknown. As already discussed, the IUknown is an interface that supports three methods; QueryInterface, AddRef, and Release.

All COM interfaces are inherited either directly or indirectly from the IUnknown, hence all other interfaces have these three functions also. With a pointer to the IUnknown, a client can get a pointer to other interfaces the object supports through QueryInterface. In short, an object can use QueryInterface to find out the capabilities of another object. If the object supports the interface, it returns a pointer to the interface. Listing 5.1 demonstrates the use of the pointer to a control's IUnknown interface to QueryInterface to find out the class information using MFC.

Listing 5.1. The use of the pointer to a control's IUnknown interface and then utilizing QueryInterface to get the class information.

```
1:  // Function to get a pointer to a control's IUnknown and use
```

```
2:  // QueryInterface to see if it supports the interface.
3:  int MyClass::DoControlWork()
4:  {
5:     LPUNKNOWN lpUnknown;
6:     LPPPROVIDECLASSINFO lpClassInfo;
7:
8:   lpUnknown = GetControlUnknown();
9:
10:     if(lpUnknown == NULL)
11:     {
12:          // return my error code to let me know IUnknown was NULL
13:          return ERROR_CODE_IUNKNOWN_NULL;
14:     }
15:     else
16:     {
17:          if(SUCCEEDED(lpUnknown->QueryInterface(IID_IProvideClassInfo,
18:                                                 (void**) &lpClassInfo)))
19:          {
20:               // QueryInterface Returned a Succeeded so this
21:               // Interface is Supported
22:               // {
23:               //          Perform some function with lpClassInfo such
24:               // as getting the class info and examining the class attributes
25:               // {
26:
27:               lpClassInfo->Release();
```

```
28:          }
29:          else
30:          {
31:               // Control Does Not Support Interface
32:               return ERROR_INTERFACE_NOT_SUPPORTED;
33:          }
34:     }
35:     return SUCCESSFUL;
36:  }
```

In addition, the object can manage its own lifetime through the AddRef and Release functions. If an object obtains a pointer to an object, then AddRef is called, incrementing the object's reference count. Once an object no longer needs the pointer to the interface, Release is called, decrementing the object's reference count. Once the reference count reaches zero, then an object can safely destroy itself.

Although the IUnknown is a must implement, you should also take a look at the other interfaces an ActiveX Control might want to implement. Table 5.2 shows the potential COM interfaces an ActiveX Control may want to support.

In addition, the control may want to implement its own custom interfaces. By implementing only the interfaces it needs, the ActiveX Control can be as lean as possible. The previous OLE Control standard required that in order to be compliant with the standard, the control had to implement certain interfaces. With ActiveX Controls, this is no longer the case; you are only required to implement IUnknown.

Table 5.2. The potential COM interfaces for an ActiveX Control.

Interface Purpose IOleObject Principal mechanism by which a control communicates with its container. IOleInPlaceObject Means by which activation and deactivation of an object is managed. IOleInPlaceActiveObject Provides communications between an in-place active object and the outermost windows of the container. IOleControl This interface allows support for keyboard mnemonics, properties, and events. IDataObject Allows for the transfer of data and the communication of changes in the data. IViewObject Allows the object to display itself. IViewObject2 An extension of the IViewObject interface. It allows you to find the size of the object in a given view. IDispatch Is an interface that can call virtually any other COM interface. It is used in OLE Automation to evoke late binding to properties and methods of COM objects. IConnectionPointContainer This interface supports connection points for connectable objects. IProvideClassInfo This interface encapsulates a single method by which to get all of the information about an object's co-class entry in its type library. IProvideClassInfo2 This interface is an extension to the IProvideClassInfo to provide quick access to an object's IID for its event set. ISpecifyPropertyPages An interface that denotes an object as supporting property pages. IPerPropertyBrowsing This interface supports methods to get access to the information in the property pages supported by an object. IPersistStream This interface provides methods for loading and storing simple streams. IPersistStreamInit Designed as a replacement for IPersistStream. Adds an initialization method InitNew. IPersistMemory Allows the method to access a fixed sized memory block for an IPersistStream object. IPersistStorage This interface supports the manipulation of storage objects to include loading, saving, and exchanging. IPersistMoniker An interface to expose to asynchronous objects the ability to manipulate the way they bind data to the object. IPersistPropertyBag This interface allows the storage of persistent properties. IOleCache An interface to control access to the cache inside an object. IOleCache2 An interface that allows the selective update of an object's cache.

IExternalConnection An interface that allows the tracking of external locking on an embedded object. IRunnableObject An interface that enables a container to control its executable objects.

Be Self-Registering

In order for an ActiveX Control, or any other COM object, to be utilized, it must be registered in the System Registry. The System Registry is a database of configuration information divided into a hierarchical tree. This tree consists of three levels of information: Hives, Keys, and Values. The system registry is a centralized place where you can go to find out information about an object (see Figure 5.9).

Note

The System Registry in Windows 95 can be viewed through a program called regedit.exe. This program can be found in the \WINDOWS directory of Windows 95 and the \WINNT\SYSTEM32 directory in Windows NT 4.0. If you are using Windows NT 3.51, the System Registry can be viewed with a program called regedit32.exe which is found in the same directory as specified for Windows NT 4.0 above.

Figure 5.9. The Windows 95 System Registry as seen through the Regedit program.

If the control is not registered in the registry, then it is unknown, and therefore unusable, by the system.

Thus, it is a requirement for ActiveX Controls to be self-registering. This means an ActiveX Control must implement and export the functions DllRegisterServer and DllUnregisterServer. In addition, it is a requirement for ActiveX Controls to register all of the standard registry entries for automation servers and embeddable objects. Listing 5.2 demonstrates the use of DllRegisterServer to support self-registration of the control using MFC. This code is generated for you by Visual C++'s Control Wizard.

Listing 5.2. Using the DllRegisterServer to support self-registration of the control.

```
1:  ////////////////////////////////////////////////////////////////////

2:  // DllRegisterServer - Adds entries to the system registry

3:

4:  STDAPI DllRegisterServer(void)

5:  {

6:        AFX_MANAGE_STATE(_afxModuleAddrThis);

7:

8:     if (!AfxOleRegisterTypeLib(AfxGetInstanceHandle(), _tlid))

9:           return ResultFromScode(SELFREG_E_TYPELIB);

10:
```

```
11:     if (!COleObjectFactoryEx::UpdateRegistryAll(TRUE))
12:          return ResultFromScode(SELFREG_E_CLASS);
13:
14:     return NOERROR;
15:  }
```

Listing 5.3 demonstrates the use of DllUnregisterServer to support self-unregistration of a control using MFC. This code is generated for you by Visual C++'s Control Wizard.

Listing 5.3. Using DllUnregisterServer to support self-unregistration of a control.

```
1:  ////////////////////////////////////////////////////////////////////
2:  // DllUnregisterServer - Removes entries from the system registry
3:
4:  STDAPI DllUnregisterServer(void)
5:  {
6:       AFX_MANAGE_STATE(_afxModuleAddrThis);
7:
8:       if (!AfxOleUnregisterTypeLib(_tlid))
9:            return ResultFromScode(SELFREG_E_TYPELIB);
10:
11:     if (!COleObjectFactoryEx::UpdateRegistryAll(FALSE))
12:          return ResultFromScode(SELFREG_E_CLASS);
13:
14:     return NOERROR;
16:  }
```

Listings 5.2 and 5.3 show how you support registration and unregistration, and Listing 5.4 shows how you register your control and its capabilities. Notice in Line 15 of Listing 5.4 the variable dwMyControlOleMisc. It contains the status bits of your control. This iv very important because it contains the capabilities of your control. These capabilities can be looked up in the System Registry to find out what capabilities your control contains without instantiating the object.

Listing 5.4. How to register your control and your controls capabilities in MFC.

```
1:  ////////////////////////////////////////////////////////////////////
2:  // CMyCtrl::CMyCtrlFactory::UpdateRegistry -
3:  // Adds or removes system registry entries for CMyCtrl
4:  BOOL CMyCtrl::CMyCtrlFactory::UpdateRegistry(BOOL bRegister)
5:  {
6:       if (bRegister)
7:            return AfxOleRegisterControlClass(
8:               AfxGetInstanceHandle(),
9:               m_clsid,        // Records the Object's CLSID
10:               m_lpszProgID,  // Records a Unique Program ID for MyControl
11:               IDS_MYCONTROL, // Records a Human Readable Name of MyControl
12:               IDB_MYCONTROL, // Recordes the Bitmap to Represent MyControl
13:               TRUE,          // Records that MyControl can be insertable
14:                              // in a Container's Insert Object Dialog
15:               dwMyControlOleMisc, // Records the Status bits of MyControl
16:               tlid,          // Records the Unique ID of the MyControls
17:                              // Control Class
18:               wVerMajor,     // Records the Major Version of MyControl
19:               wVerMinor);    // Recorde the Minor Version of MyControl
20:     else
21:          return AfxOleUnregisterClass(m_clsid, m_lpszProgID);
22:  }
```

The possible status bits that can be set for a control are shown in Table 5.3. These bits identify the capabilities of the control. Take a moment to become familiar with them, as they will become even more important in the discussion of OLE Containers in Chapter 7, "Microsoft Internet Explorer 3.0 and Its Scripting Object Model."

Table 5.3.The OLE miscellaneous status bits symbolic constants and what they mean to controls and objects.

Symbolic Constant Meaning OLEMISC_RECOMPOSEONRESIZE Identifies an object that upon re-sizing by the container will re-scale its presentation data. OLEMISC_ONLYICONIC Identifies an object that only exists in the iconic state. OLEMISC_INSERTNOTREPLACE Identifies an object that initializes itself from the currently selected container data. OLEMISC_STATIC Identifies that an object is static and contains no native data, only presentation data. OLEMISC_CANTLINKINSIDE This flag identifies items such as OLE 1.0 Objects, static objects, and links. These objects cannot be a linked source that when bound to runs another object. OLEMISC_CANLINKBYOLE1 Identifies that an object can be linked by the containers that conform to OLE 1.0 specification. OLEMISC_ISLINKOBJECT Identifies that an object is a linked object. This is only important for OLE1 objects. OLEMISC_INSIDEOUT Identifies that an object can be in-place activated without the need for toolbars or menus. OLEMISC_ACTIVATEWHENVISIBLE Identifies that an object can only be activated in the visible state. The OLEMISC_INSIDEOUT flag must also be set. OLEMISC_RENDERINGISDEVICEINDEPENDENT This flag identifies that the object's presentation data will remain the same regardless of the target container. OLEMISC_INVISIBLEATRUNTIME Identifies controls that are invisible at runtime, such as Internet Explorer's Timer Control or Internet Explorer's PreLoader Control. OLEMISC_ALWAYSRUN Tells a control that a control should be set in the running state even when not visible. OLEMISC_ACTSLIKEBUTTON Identifies controls that can act like buttons. OLEMISC_ACTSLIKELABEL Identifies controls that can change the label provided by the container. OLEMISC_NOUIACTIVATE Identifies whether a control supports User Interface activation. OLEMISC_ALIGNABLE This bit identifies that a control may be aligned with other controls for containers that support control alignment. OLEMISC_SIMPLEFRAME Identifies that the control supports the ISimpleFrameSite interface. OLEMISC_SETCLIENTSITEFIRST In the new OLE Container specification, this flag identifies controls that support the SetClientSide function being called after the control is created but before it is displayed. OLEMISC_IMEMODE In the Double Byte Character Set versions of Windows, identifies that the control supports the Input Method Editor Mode, for internationalized controls.

These miscellaneous status bits are especially important when used in conjunction with Component Categories as an accurate picture of what your control can or cannot do. This picture of what the control can do, can be obtained from the System Registry.

Component Categories

Previously, in order to be registered on the system, an OLE Control was registered through entries in the registry with the Control keyword. To your benefit, controls can be utilized for multiple purposes. Therefore, a way was needed to identify a control's functionality as opposed to just listing the interfaces it supports. This is where Component Categories come in.

Component Categories are a way of describing what a control does. They provide a better method for containers to find out what a control does without creating it and having to query for its methods using an IUnknown pointer and QueryInterface. Creating a control object involves a lot of overhead. A container would not want to create a control if the container itself does not support the functionality the control requires.

Component Categories are not specific to ActiveX but are an extension of the OLE Architecture. Each Component Category has its own GUID (Globally Unique Identifier) and a human readable name stored in a well-known place in the System Registry. When a control registers itself, it does so using its component category id. In addition, it also registers the component categories it supports and the component categories it requires its container to support.

For backward compatibility, the control should also register itself with the Control keyword for containers that do not support the new component categories. The control should also register the key ToolBoxBitmap32. This key identifies the module name and resource id for a 16*15 bitmap. ToolBoxBitmap32 provides a bitmap to use for the face of a toolbar or toolbox button in the container application. If a control can be inserted in a compound document, it should also register the Insertable key.

Component Categories can be mixed and matched depending on their type. Microsoft maintains a list of Component

Categories. Any categories that are new should be submitted to Microsoft for inclusion in the list. This promotes interoperability. The following component categories have been identified:

- Simple Frame Site Containment
- Simple Data Binding
- Advanced Data Binding
- Visual Basic Private Interfaces
- Internet-Aware Controls
- Windowless Controls

SimpleFrameSite Containment

A Simple Frame Site Container Control is a control that contains other controls, for example a 3D group box that contains a group of check boxes. The GUID for this component category is: CATID - {157083E0-2368-11cf-87B9-00AA006C8166} CATID_SimpleFrameControl. In order to support a Simple Frame Site Container, the OLE Container application must implement the ISimpleFrameSite interface and the control must have its status bit set to OLEMISC_SIMPLEFRAME.

Simple Data Binding

A control or container that supports simple data binding supports the IPropertyNotifySink interface. Data binding is how controls affiliate their persistent properties and how containers exchange property changes from their User Interface to the control's persistent properties. This allows the persistent storage of their properties and at runtime binds the data to the control synchronizing property changes between the control and the container. The GUID for this component category is: CATID - {157083E1-2368-11cf-87B9-00AA006C8166} CATID_PropertyNotifyControl.

Note

Although a control that supports simple data binding is meant to provide binding to a datasource, such binding should not be required for the functionality of the control. Even though a lot of the functionality the control has is lost, that control should degrade gracefully and still be able to function, although potentially limited, independent of any data binding.

Advanced Data Binding

Advanced Data Binding is similar to Simple Data Binding except it supports more advanced binding techniques, such as asynchronous binding and Visual Basic Data Binding. The GUID for this component category is: CATID - {157083E2-2368-11cf-87B9-00AA006C8166} CATID_VBDataBound.

Visual Basic Private Interfaces

These component categories are for components that specifically support the Visual Basic environment. Controls or containers may want to support alternative methods in case a container encounters a control, or a control encounters a container that does not support the Visual Basic Private Interfaces. The GUID for this component category is: CATID - {02496840-3AC4-11cf-87B9-00AA006C8166} CATID_VBFormat, if the container implements the IBVFormat interface for data formatting to specifically integrate with Visual Basic or CATID - {02496841-3AC4-11cf-87B9-00AA006C8166} CATID_VBGetControl if the container implements IVBGetControl so that controls can enumerate other controls on a Visual Basic Form.

Internet-Aware Controls

Internet-aware controls implement one or more persistent interfaces to support operation across the Internet. All these categories provide persistent storage operations. The following are GUIDs for components that fall into this category:

- CATID - {0de86a50-2baa-11cf-a229-00aa003d7352} CATID_RequiresDataPathHost
- CATID - {0de86a51-2baa-11cf-a229-00aa003d7352} CATID_PersistsToMoniker
- CATID - {0de86a52-2baa-11cf-a229-00aa003d7352} CATID_PersistsToStorage
- CATID - {0de86a53-2baa-11cf-a229-00aa003d7352} CATID_PersistsToStreamInit
- CATID - {0de86a54-2baa-11cf-a229-00aa003d7352} CATID_PersistsToStream
- CATID - {0de86a55-2baa-11cf-a229-00aa003d7352} CATID_PersistsToMemory
- CATID - {0de86a56-2baa-11cf-a229-00aa003d7352} CATID_PersistsToFile
- CATID - {0de86a57-2baa-11cf-a229-00aa003d7352} CATID_PersistsToPropertyBag

The RequiresDataPathHost category means that the object requires the container to support the IBindHost interface because the object requires the capability of saving data to one or more paths.

All of the rest of the categories listed above are mutually exclusive. They are used when an object only supports a single persistence method. If a container does not support a persistence method that a control supports, the container should not allow themselves to create controls of that type.

Windowless Controls

Windowless Controls are controls that do not implement their own window and rely on the use of their container's window to draw themselves on. These types of controls are non-rectangular controls such as arrow buttons, gauges, and other items modeled after real-world objects. In addition, this includes transparent controls. The GUID for this component category is: CATID - {1D06B600-3AE3-11cf-87B9-00AA006C8166} CATID_WindowlessObject.

Component Categories and Interoperability

Components that do not support a category should degrade gracefully. In the case where a control or container is unable to support an interface, the control should either clearly document that a particular interface is required for the proper operation of the component or at runtime notify the user of the component's degraded capability.

By using self-registration, components can be self-contained, which is necessary for Internet operations. By using

DllRegisterServer and DllUnregisterServer and the Component Categories API functions to register itself and the component categories it supports, a control can further its interoperability in a variety of environments.

Code Signing

In the Internet environment, users must download the components to their local machine and utilize them. This is an extreme hazard to the local machine by allowing the implementation of this foreign code on their machine.

This is where a new security measure called code signing comes in. Browsers typically warn the user that they are downloading a potentially unsafe object; however, it does not physically check the code for authenticity to ensure it has not been tampered with nor does it verify its source.

Microsoft has implemented Authenticode, which embodies the Crypto API. This allows developers to digitally sign their code so that it can be checked and verified at runtime. This function is built into the browser and displays a certificate of authenticity (see Figure 5.10) if the control is verified.

Figure 5.10. The certificate the user is shown at runtime after the code has been authenticated.

Presently, the code signing specification and the certification process are being reviewed by the World Wide Web Consortium (W3) and the current specifications are subject to change. Internet Explorer and all Microsoft controls naturally support code signing and Authenticode, but as of yet Netscape does not. Netscape has gone to W3 with a proposal to extend its own "digital certificate" standard. In the spirit of cooperation, Netscape eventually will support the code signing specification, or at a minimum Microsoft will embrace both standards.

Code signing works with DLLs, EXEs, CABs, and OCXs. When a developer creates these items, they attain a digital certificate from an independent Certification Authority. They then run a one-way hash on the code and produce a digest that has a fixed length. Next, the developer encrypts the digest using a private key. This combination of an encrypted digest coupled with the developer's certificate and credentials is a signature block unique for the item and the developer. This signature block is embedded into the executable program.

Here's the way code signing works on the client machine. When a user downloads a control, for example, from the Internet, the Browser application such as Internet Explorer or Netscape calls a WIN32 API function called WinVerifyTrust.

Note

At present Netscape does not currently support code signing.

WinVerifyTrust then reads the signature block. With the signature block, the WinVerifyTrust can authenticate the certificate and decrypts the digest using the developer's public key. Using the public key, the function then rehashes the code with the hash function stored in the signature block and creates a second digest. This digest is then compared with the original. If they do not match, this indicates tampering and the user is warned (see Figure 5.11). On the contrary if the digest had matched, instead of the warning in Figure 5.11, the user would have gotten the certificate of authenticity as seen in Figure 5.10.

Figure 5.11. The warning the user is shown at runtime to tell the user of a potential danger because the code cannot be authenticated.

Despite Code Signing, the user is in control and may choose to heed or ignore the warning. If the hashes check out, then a certificate is displayed by the browser.

The Code Signing mechanism provides some security for end users and developers alike. It is a deterrent to malicious

tampering with executable code for the intent of information warfare such as viruses, and it is also a deterrent for those who might pirate the code developed by others. Please be aware again that this is a proposed standard and has not yet been officially accepted, although there is nothing I can see at this time that can compete with it. It is safe to say that no matter what Microsoft will continue to support it and in addition, continue to refine it. The bottom line is you will need to continue to monitor the standard.

Performance Considerations

ActiveX Controls are designed to work across the Internet. As such they are Internet-aware. Unfortunately, the Internet is low bandwidth and highly subjected to server latency. This means that ActiveX Controls must be lean and mean, or to put it more plainly, highly optimized. Because ActiveX Controls implement only the interfaces they need, they are already partially optimized. ActiveX Controls are optimized to perform specific tasks. However, there are several things you can do to help optimize your controls.

- Optimize control drawing
- Don't always activate when visible
- Provide flicker-free activation
- Provide windowless activation
- Optimizing persistence and initialization
- Use windowless controls
- Use a device context that is unclipped
- While inactive, provide mouse interaction

Tip

These performance considerations and optimizing techniques apply to OLE Controls as well as ActiveX Controls. You may have already developed OLE Controls to the old standard, but you can still apply most of these principles to those controls.

Optimize Control Drawing

When you draw items, you have to select items such as pens, brushes, and fonts into the device context to render an object on the screen. Selecting these into the device context requires time and is a waste of resources when the container has multiple controls that are selecting and deselecting the same resources every time they paint. The container may support optimized drawing. This means that the container handles the restoration of the original objects after all the items have been drawn. IViewObject::Draw supports optimized drawing by using the DVASPECTINFOFLAG flags set in DVASPECTINFO structure. You must use this to determine if your container supports optimized drawing when implementing API functions. MFC encapsulates this check for you in the COleControl::IsOptimizedDraw function. You can then optimize how you draw your code by storing your GDI objects as member variables instead of local variables. This prevents them from being destroyed when the drawing function finishes. Then if the container supports optimized drawing, you do not need to select the objects back because the container has taken care of this for you.

Don't Always Activate When Visible

If your control has a window, it may not need to be activated when visible. Creating a window is a control's single biggest operation and therefore should not be done until it is absolutely necessary. Therefore, if there is no reason for your control to be activated when visible, you will need to turn off the OLEMISC_ACTIVATEWHENVISIBLE miscellaneous status bit.

Provide Flicker-Free Activation

When your control has a window, it must sometimes transition from the active to the inactive state. There is a visual flicker that occurs when the control redraws from the active to the inactive state. Flicker can be eliminated by two methods; drawing off-screen and copying to the screen in one big chunk, and drawing front to back. IViewObjectEx API function provides the necessary functions to use either method or a combination of both. With MFC the implementation is much simpler. (See Listing 5.5.)

Listing 5.5. Shows how you set the windowless flag in MFC.

```
1:  DWORD CMyControl::GetControlFlags()
2:  {
3:     return COleControl::GetControlFlags() | noFlickerActivate;
4:  }
```

Optimizing Persistence and Initialization

Optimizing persistence and initialization means basically one thing: Keep your code as lean as possible. Because of the cheapness of hard drive space and memory, some programmers have gotten lazy in the creation of this code and allowed it to become bloated and slow. With Internet applications, this is a death sentence. Most people access the Internet with 14.4 modems. A megabyte of data takes almost 9 minutes on a 14.4 modem. Users will get impatient if they have to wait long periods of time. What can you do? You can do several things.

First of all, make sure you do not leave any non-utilized blocks of code or variables. You should also take out any debugging or testing blocks out of your code. For example, you have written your code so a message box displays when you reach a certain segment of code. Take it out! It will only add to your code size. However, if you delimit your debugging blocks of code using the preprocessor #ifdef _DEBUG and #endif you will not have to worry about the code being included in the release builds, as the debugging blocks of code will be left out of the compile.

Second, today's compilers have optimizing options on them. In the past these optimizing compilers were not very efficient and sometimes introduced bugs in an application that had already been tested. But, compilers have gotten much better. Use them! Let the compiler do some of the work for you. You may have to tweak and play with the optimizations to find the best combination of options.

Warning

Make sure you perform your compiler optimizations before you send your code to testing. However, any time you touch the code, it should go back through testing. Therefore, if you should have to tweak the compiler optimizations after it has

been through testing. Make sure you send it back through testing! This may help prevent discovering a bug after release.

You should also turn off the incremental linking option on your compiler when you do a release build. Incremental linking can add serious bloat to your code.

Note

For an excellent article on keeping your code small, see "Removing Fatty Deposits from Your Applications Using This 32-bit Liposuction Tools" by Matt Pietrek in *Microsoft Systems Journal*, October 1996, Vol 11, No 10. Matt Pietrek has many useful suggestions and even provides a nice tool to assist you.

The last thing you should take into account is utilizing asynchronous operations to perform initialization and persistence operations. Asynchronous downloading gives the user the illusion that things are occurring faster than they are. In addition, you may want to give the user other visual cues that progress is being made, such as a progress indicator or a message box. However, you will have to weigh the performance issues associated with their addition.

Use Windowless Controls

You should consider making your control a Windowless Control if appropriate. Creating a window is a control's single biggest operation, taking almost two-thirds of its creation time. This is a lot of unnecessary overhead for the control. Most of the time, a control does not need a window and can utilize its container's window and allow the container to take on the overhead of maintaining that window. This will allow you to model your controls after real-world objects, such as gauges, knobs, and other non-rectangular items.

By using the API function IOleInPlaceSiteEx::OnInPlaceActivateEx and setting the ACTIVATE_WINDOWLESS flag, you can have your control be in the windowless mode. With MFC, you can do the following:

```
DWORD CMyControl::GetControlFlags()

{

    return COleControl::GetControlFlags() | windowlessActivate;

}
```

In addition, there is a whole series of API functions that allow you to manipulate windowless controls. MFC has encapsulated many of these functions for you also. The books online in Visual C++ has a complete reference for these functions. In addition, the Win 32 API references have the API level functions.

Use a Device Context that is Unclipped

If you have a window and you are sure your control does not draw outside of that window, you can disable the clipping in your drawing of the control. This can yield a small performance gain by not clipping the device context. With MFC, you can do the following to remove the clipPaintDC flag:

```
DWORD CMyControl::GetControlFlags()

{

     return COleControl::GetControlFlags() & ~clipPaintDC;

}
```

Note

The clipPaintDC flag has no effect if you have set your control as a windowless control.

With the API functions in the ActiveX SDK, you can implement the IViewObject, IViewObject2, and IViewObjectEx interfaces to optimize your drawing code so you do not clip the device context.

While Inactive, Provide Mouse Interaction

You may set your control to inactive because it does not always need to be activated when visible. You may still want your control to process mouse messages such as WM_MOUSEMOVE and WM_SETCURSOR. You will need to implement the IPointerInactive interface to allow you to process the mouse messages. If you are using MFC, you need only implement the following function as the framework handles the rest for you.

```
DWORD CMyControl::GetControlFlags()

{

     return COleControl::GetControlFlages() | pointerInactivate;

}
```

However, you will need to override the OLEMISC_ACTIVATEWHENVISIBLE miscellaneous status bit with OLEMISC_IGNOREACTIVATEWHENVISIBLE. This is because the OLEMISC_ACTIVATEWHENVISIBLE forces the control to always be activated when visible. You have to do this to prevent the flag from taking effect for containers that do not support the IPointerInactive interface.

Reinventing the Wheel

In today's software development environment, software engineers are not only designers and programmers, but increasingly, software engineers are taking on the role of component integrators. End users demand that their software be developed quickly, be rich in features, and integrate with the rest of the software they use. With the advent of OLE, CORBA, and OpenDoc, you now have hundreds of thousands of reusable components and objects to choose from. There is an abundance of dynamic link libraries, controls, automation components, and document objects at your fingertips. OLE Controls especially provide an off-the-shelf self-contained reusable package of functionality, created by someone else. OLE Controls provide functionality of all types such as multimedia, communications, user interface components, report writing, and computational (see Figure 5.12).

Figure 5.12. Some of the numerous OLE/ActiveX Controls available.

This is functionality that you do not have to create. The major key to component integration is to be able to integrate all of the components with a custom application so that they work in single harmonious union as if they were native to the application.

However, before you embark on creating this application, you should take care not to "reinvent the wheel." OLE/ActiveX Controls, the Component Object Model, and the object-oriented paradigm present a unique opportunity for you to truly have code reuse. In order to achieve this nirvana of code reuse, you should evaluate what components are already out there. Likewise, before you decide to write your own OLE Controls, you should take a look at what is already out there and see if you can utilize what is already available as opposed to reinventing the wheel.

When you choose to utilize off-the-shelf components, there are a few things you should consider. You should ask the following questions: How long has the manufacturer been in business? does the company supply the source code with the component? (The source code would come in handy if the manufacturer went out of business or had a bug in its component that it was not going to fix.) what are the licensing fees and distribution costs? is the company web enabled? what kind of support and money-back guarantee does the manufacturer provide? what tools will the component be supported in?

These questions can save you a lot of heartache later. Integration of these off-the shelf-components is sometimes tricky. Make sure you thoroughly research the components you choose.

Internet Explorer Stock Controls

So that you do not go out and reinvent the wheel, it is important to note that there are several controls that come stock with Internet Explorer. You can utilize these controls in your Web pages and in your application development efforts. These controls provide a variety of functionality. The following ActiveX Controls come with Internet Explorer:

- Animated button control
- Chart control
- Gradient control
- Label control
- Marquee control
- Menu control
- PopUp menu control
- [1b] PopUp window control
- Stock ticker control
- PreLoader control
- Timer control
- View tracker control

Note

A demo of the functionality of each of these controls is available on the Microsoft World Wide Web site at the following Internet URL: http://www.microsoft.com/activex/gallery/default.htm. In addition, a number of other third-party vendors have their controls demonstrated at the same Microsoft WWW Site.

Animated Button

The Animated Button ActiveX Control displays frame sequences of an AVI file using the Microsoft Windows Animation common control, based on the state of the button.

Chart

The Chart ActiveX Control allows you to display a variety of charts such as a bar chart, pie charts, and graphs.

Gradient

The Gradient ActiveX Control allows you to display a gradient of one palettized color to another, gradually fading the pixels from one color to another.

Label

The Label ActiveX Control allows you to display text at various angles, sizes, and colors. It will even allow you to display text around a user-defined curve.

Marquee

The Marquee ActiveX Control allows you to have scrolling, bouncing, or sliding text and URLs within a window, much like the old cinema marquees.

Menu

The Menu ActiveX Control allows you to embed menu button or pull-down menu functionality in your Web page.

Popup Menu

The Popup Menu ActiveX Control allows you to embed a popup menu in your Web page. This control sends a Click event when the user selects a menu item.

Popup Window

The Popup window control enables you to display a specified HTML document in a popup window. In addition, this control can be used to provide tooltips or preview links.

Preloader

The Preloader ActiveX Control holds the position of a URL and stores it in cache. Once it is activated, it downloads asynchronously in the background the item pointed to by the URL. This control is not visible at runtime.

Stock Ticker

The Stock Ticker ActiveX Control acts just like a stock ticker and displays data across the screen at a set speed. It utilizes text files or XRT files that are downloaded asynchronously at specified intervals.

Timer

The Timer ActiveX Control fires an event periodically at a set time interval. The timer control is invisible at runtime.

View Tracker

The View Tracker ActiveX Control fires OnShow and OnHide events based on whether the control is in or out of the viewable area of the screen.

As you can see, there are several controls provided in Internet Explorer that have a lot of useful capability built into them. Be aware of what is already out there, and it may save you development time when every minute counts.

Displaying a Control in a Web Page

In order to "Activate the Internet" with ActiveX Controls, as the Microsoft Marketing folks are fond of saying, you have to have a way of embedding those ActiveX Controls in an HTML file.

The World Wide Web Consortium (W3C) controls the HTML standard. The current HTML standard is version 3.2. Like most standards, it is continually updated and modified as technology progresses. As the standard progresses, the controlling agency tries to ensure backward compatibility. This is so that any HTML browser that does not yet support the newest standard will degrade gracefully and allow the HTML to be viewed.

Note

The current World Wide Web Consortium (W3C) HTML standard is available at the following Internet URL: http://www.w3.org/pub/WWW/.

The <OBJECT> HTML tag is used to allow the insertion of dynamic content in the Web page such as ActiveX Controls. The tag is just a way of identifying such dynamic elements. It is up to the browser to parse the HTML tags and perform the appropriate action based on the meaning of the tag. In Listing 5.6, you can see the HTML syntax for the <OBJECT> tag. This syntax comes directly from the World Wide Web Consortium (W3C) controls HTML standard Version 3.2.

Listing 5.6. The HTML syntax for the <OBJECT> tag.

```
1:  <OBJECT
2:      ALIGN= alignment type
3:      BORDER= number
4:      CLASSID= universal resource locator
5:      CODEBASE= universal resource locator
6:      CODETYPE= codetype
7:      DATA= universal resource locator
8:      DECLARE
9:      HEIGHT= number
10:     HSPACE= value
11:     NAME= universal resource locator
12:      SHAPES
13:      STANDBY= message
14:      TYPE= type
15:      USEMAP= universal resource locator
16:      VSPACE= number
17:      WIDTH= number
18: </OBJECT>
```

By utilizing the <OBJECT> tag, you can insert an object such as an image, document, applet or control, into the HTML document.

Table 5.4 shows the acceptable range of values to be utilized by the parameters of the <OBJECT> tag.

Table 5.4. The values for the parameters of the <OBJECT> tag.

Parameter Values ALIGN= alignment type Sets the alignment for the object. The alignment type is one of the following values: BASELINE, LEFT, MIDDLE, CENTER, RIGHT, TEXTMIDDLE, TEXTTOP, and TEXTBOTTOM. BORDER= number Specifies the width of the border if the object is defined to be a hyperlink. CLASSID= universal resource locator Identifies the object implementation. The syntax of the universal resource locator depends on the object type. For example, for registered ActiveX controls, the syntax is: CLSID:*class-identifier*. CODEBASE= universal resource locator Identifies the codebase for the object. The syntax of the universal resource locator depends on the object. CODETYPE= codetype Specifies the Internet media type for code. DATA= universal resource locator Identifies data for the object. The syntax of the universal resource locator depends on the object. DECLARE Declares the object without instantiating it. Use this when creating cross-references to the object later in the document or when using the object as a parameter in another object. HEIGHT= number Specifies the height for the object. HSPACE= number Specifies the horizontal gutter. This is the extra, empty space between the object and any text or images to the left or right of the object. NAME= universal resource locator Sets the name of the object when submitted as part of a form. SHAPES Specifies that the object has shaped hyperlinks. STANDBY= message Sets a message to be displayed while an object is loaded. TYPE= type Specifies the Internet media type for data. USEMAP= universal resource locator Specifies the image map to use with the object. VSPACE= number Specifies a vertical gutter. This is the extra white space between the object and any text or images above or below the object. WIDTH= number Specifies the width for the object.

In Listing 5.7, you can see HTML document source code with an embedded ActiveX object in it. In addition, notice the <PARAM NAME= *value*> tag. This tag was utilized to set any properties your ActiveX Control may have.

Listing 5.7. The HTML Page with an embedded <OBJECT> tag showing an ActiveX ActiveMovie Control embedded in the page.

```
1:  <HTML>
2:  <HEAD>
3:  <TITLE>AN EMMBEDDED ActiveX Control</TITLE>
4:  </HEAD>
5:  <BODY>
6:
7:  <p align=center><font size=6><em><strong><u>An EMMBEDDED ActiveX Control
    </u></strong></em></font></p>
8:  <OBJECT
9:  ID="ActiveMovie1"
10: WIDTH=347
11: HEIGHT=324
12: ALIGN=center
```

```
13: CLASSID="CLSID:05589FA1-C356-11CE-BF01-00AA0055595A"
14: CODEBASE="http://www.microsoft.com/ie/download/activex/amovie.ocx#
    Version=4,70,0,1086"
15:     <PARAM NAME="_ExtentX" VALUE="9155">
16:     <PARAM NAME="_ExtentY" VALUE="8573">
17:     <PARAM NAME="MovieWindowSize" VALUE="2">
18:     <PARAM NAME="MovieWindowWidth" VALUE="342">
19:     <PARAM NAME="MovieWindowHeight" VALUE="243">
20:     <PARAM NAME="FileName" VALUE="E:\vinman\duds.avi">
21:     <PARAM NAME="Auto Start" VALUE="TRUE">
22: </OBJECT>
23:
24: </BODY>
25: </HTML>
```

When a browser such as Internet Explorer encounters this page, it begins to parse the HTML source code. When it finds the <OBJECT> in line 8, it realizes it has encountered a dynamic object. The browser then takes lines 10-12, the WIDTH, HEIGHT, and ALIGN attributes, which are in this case are 347, 324, and CENTER respectively, and sets up a placeholder for the object on the rendered page. It then takes the ID "ActiveMovie1" in line 9 and the CLASSID "CLSID:05589FA1-C356-11CE-BF01-00AA0055595A" in line 13 and checks to see whether this control has been registered before in the registry. If the control object has never been registered, it then uses the CODEBASE attribute to locate the OCX on the server machine and proceeds to download the object into the \Windows\Ocache directory. The browser then registers the AMOVIE.OCX by calling the function DllRegisterServer to register the control on the local machine. Now with the control properly registered, the browser can get the CLSID for the object from the registry. In order to utilize the control, it passes the CSLID to CoCreateInstance to create the object, and this returns the pointer to the control's IUnknown. It can utilize this pointer and the property information in lines 15-22 to actually render the object on the page. Figure 5.13 shows the HTML document displayed in Internet Explorer.

Figure 5.13. The HTML document as it appears in Internet Explorer with the ActiveX ActiveMovie Control embedded in it.

Now you can see that embedding controls to enhance a Web page with dynamic content is fairly easy. It is important that you as an ActiveX Control designer understand how they are rendered. A more in-depth look at creating Web pages and using controls and applets in them will be presented in Chapter 12, "Advanced Web Page Creation."

ActiveX Control Pad

The ActiveX Control Pad is discussed in full in Chapter 11, "Using ActiveX Control Pad." It provides a method of generating the HTML code that was discussed earlier, to embed ActiveX and other dynamic objects into HTML source (see Figure 5.14). This is a free tool provided by Microsoft to aid in the production of Internet-enabled applications.

Note

The ActiveX Control Pad can be downloaded from Microsoft at the following Internet URL: http://www.microsoft.com/workshop/author/cpad.

This tool can be used to quickly embed your control in a page so you can test its functionality. The ActiveX Control Pad can be a great timesaver, freeing you from having to remember how to write HTML source code. It will even allow you to test your ability to utilize VBScript (see Figure 5.15) to do OLE Automation with your code.

Figure 5.14. The ActiveX Control Pad with the Active Movie Control properties being edited.

Figure 5.15. The ActiveX Script Wizard to help you create scripts to further "Activate" your controls.

In addition, the ActiveX Control Pad comes with a suite of ActiveX Controls for you to utilize in the development of your Web pages and your OLE-enabled applications. Some of these controls are the same controls that come with Internet Explorer; however, there are a few new ones to add to your bag of OLE Controls.

OLE Controls in Development Tools

One last place to look for OLE Controls is in your development tools. Visual C++, Borland C++, Visual Basic, Delphi, PowerBuilder, Access, and almost any other mainstream Windows or Internet development suite come with OLE Controls nowadays. Become familiar with the development tools you use and take advantage of the components provided for you. This will make your job much easier and make your users much happier.

Methods of Creating OLE (ActiveX) Controls

This section examines the ways to write ActiveX Controls. Each method is discussed, and you will create an example OLE control. Presently, there are three ways of creating ActiveX Controls:

- Visual C++ and Microsoft Foundation Classes (MFC)
- ActiveX Template Library (ATL)
- ActiveX Development Kit (BaseCtl Framework)

Every major PC C++ compiler manufacturer now supports MFC, so it should be possible to create an ActiveX Control with another vendor's product; however, for the purposes of this chapter, you will do this example utilizing Visual C++ 4.2a or greater. In addition, you should be able to use any C++/C compiler to use the ActiveX Development Kit and the ActiveX Template Library.

The traditionally used programming language for creating OLE Controls is C++ and C; however, Microsoft has promised a compiled version of Visual Basic, called Visual Basic 5.0, that will be able to create OLE Controls. It has been rumored that Microsoft is also creating a converter that will convert ActiveX Controls into Java applets. Those of you who are Borland Delphi PASCAL programmers can now create ActiveX Controls with a third-party add on from Apiary Inc. called OCX Expert. This Delphi add-on takes VCLs created in Delphi and converts them to 32-bit ActiveX Controls.

Note

Information on OCX Expert can be obtained from Apiary Incorporated at the following Internet URL: http://www.apiary.com/.

Visual Basic is a very popular language/development tool because it is very easy to learn and utilize. It has even been called a quasi-4GL. Many times it is chosen for development endeavors for these reasons. However, C++ provides much more power and flexibility in the development of applications as a whole, but most importantly, in the development of user interface elements like OLE Controls.

Developers are often almost fanatically religious about their development tools. As a software engineer, you should be concerned with the right tool to fit the job. I highly recommends Visual C++ and Microsoft Foundation Classes. Programming in C++ is now much easier with class libraries such as MFC and Integrated Development Environments such as Visual C++. Some even consider it a 4GL, but theoreticians might argue that point. Nevertheless, it is a very powerful tool, one you should consider using. There are also a number of powerful C++ development tools such as Borland C++, Symantec C++, and IBM Visual Age. There is a virtual plethora of tools for you to choose from. In addition, Borland Delphi is a powerful tool with which you can develop Windows applications using Object-Oriented PASCAL, but its principal strength is its integrated development environment. If you are unfamiliar with some of these tools, you might want to get an evaluation copy of them and try them. This will better aid you in picking the right tool for the job.

This book covers OCXs created with C++. The authors want to make you aware that there are many new emerging technologies and product associated with ActiveX Controls.

The next sections examine three ways of creating ActiveX Controls. These sections assume that you have a working knowledge of C++ and a Windows class library such as Microsoft Foundation Classes (MFC) or Borland's Object Windows Library (OWL). They make no attempt to teach you C++ or MFC. Although these sections are for intermediate to advanced programmers, if you are new or curious to or about C++, MFC, or Visual C++, some information is highlighted to aid you.

Take a look now at an example of an OLE Control created with Visual C++ and MFC.

The Visual C++ and MFC Way

Visual C++ and MFC comes in three flavors, 16-bit, Win32s/32-bit, and 32-bit. Visual C++ 1.52c and MFC 2.53 for 16-bit developers and Visual C++ 4.2 and MFC 4.2 for 32-bit developers. For those of you who still desire the Win32s development platform for the development of 32-bit applications to run under 16-bit Windows, there are Visual C++ 4.1 and MFC 4.1. The newer versions of Visual C++ will no longer support Win32s. This section concentrates on the 32-bit environment and does not cover Win32s or 16-bit development. Building 16-bit OLE Controls is possible with Visual C++ 1.52c, but 16-bit development is rapidly being left behind. In addition, the statement about 16-bit development being left behind can be said of the Win32s world as well.

Obtaining the ActiveX SDK

First, you need to get the ActiveX Software Development Kit. An updated version of the ActiveX SDK was released at the same time Internet Explorer 3.0 was released. The new ActiveX SDK was updated to include the new technology and features of Internet Explorer 3.0.

Note

The most current ActiveX SDK is available from Microsoft, at no charge, at the following Internet URL: http://www.microsoft.com/intdev/sdk/.

Caution

The ActiveX SDK is only intended to run on Windows 95 and Windows NT 4.0 (release) machines running the release version of Internet Explorer 3.0.

The ActiveX SDK file obtained from the Microsoft WWW site is a self-extracting archive. In addition, if you subscribe to Level II or higher of the Microsoft Developers Network Library, the most recent ActiveX SDK should be included in future releases of MSDN.

Tip

If you do not already subscribe to the Microsoft Developers Network Library (MSDN), the authors highly recommend you do. It is an invaluable source of technical information for developers of software and hardware for the Windows Family of Operating Systems. You can obtain information on MSDN by calling Microsoft at 1-800-759-5474 or at the Microsoft WWW site at the following Internet URL: http://www.microsoft.com/msdn/.

The Microsoft Developers Network is a subscription for four levels of information and products. It contains, depending on what level you subscribe to: all the Software Development Kits, all the Knowledge Bases, documentation for all of Microsoft's developer products, how-to articles, samples, bug lists and workarounds, all the operating systems, specifications, Device Driver Kits, and the latest breaking developer news. It is issued in CD format (see Figure 5.16) and is released and updated quarterly. The Level II subscription alone comes with over 35 CDs, packed full of development information that is updated quarterly. The MSDN Library CD directly integrates with the Visual C++ IDE.

Figure 5.16. The Microsoft Developers Network Library CD.

Using the Right Version of Visual C++ and MFC

Visual C++ comes as a yearly subscription, or you can purchase the single release professional version. The professional version is version 4.0. With the subscription, you get the updates throughout the year. Right now that is version 4.2. You can only get 4.2 through the subscription. You can still create OLE Controls with version 4.0, but to get the enhancements to create ActiveX Controls you must have version 4.2. In addition, Microsoft recently released Visual C++ Enterprise Edition 4.2. The Enterprise edition of Visual C++ includes additional database tools such as an SQL debugger and visual database views. The Enterprise Edition 4.2 or greater can be utilized to develop ActiveX Controls.

As previously mentioned, a new version of the ActiveX SDK was just released in September 1996. The Visual C++ development team at Microsoft has also released a Patch for Visual C++ 4.2 and MFC 4.2 to allow developers to utilize the new features to create ActiveX applications. This patch will be included in the next subscription release of Visual C++ and MFC 4.3. The patch is called Visual C++ Patch 4.2b. You will need to download this patch and incorporate it with Visual C++ 4.2 in order to create ActiveX Controls.

Warning

The Visual C++ 4.2b Release is only for use on Visual C++ and MFC versions 4.2. Do not apply this patch to any other version of Visual C++, or your software and operating system may not operate properly.

Note

The Visual C++ 4.2b Release patch is available from Microsoft, at the following Internet URL: http://www.microsoft.com/visualc/v42/v42tech/v42b/vc42b.htm.

The patch is a self-extracting archive. Once you get the patch and extract it, make sure you follow the directions in the readme file. (If you are reading this and Visual C++ and MFC 4.3 have been released and you have them loaded, then you need not apply the patch.)

Using Visual C++ and MFC for ActiveX and OLE Controls

Previously OLE Controls had to have certain interfaces implemented whether they needed them or not. This means that controls were larger than they needed to be. This is fine if you are utilizing them on a local machine, but with ActiveX Controls that need to be downloaded and installed across the low bandwidth, high latency Internet, any excess baggage is less efficient in achieving this end. In order to get Web Masters to utilize your controls to activate their Web pages, your ActiveX Controls need to be lean, mean, efficient downloading machines.

Visual C++ comes with a Control Wizard to help you create controls. It is one of the fastest ways to create a control. In fact, if you are a newcomer to creating controls, it is the best way to learn. Why? Because it creates a framework for you. You can be up and running very quickly. However, there are a few drawbacks you need to be aware of.

In order to utilize a control created with Visual C++ and based on MFC, the MFC dynamic link library (DLL) must reside on the client machine. This file is about 1.2M and must be downloaded to the client machine. However, this must only occur the first time, if the MFC DLL does not reside on the client machine already. So you take a small performance hit the first time your control is used. Furthermore, it should also be noted that MFC-based controls tend to be fatter than the controls created by the other two methods.

You will need to weigh the options carefully, considering performance, programmer skill, timetable, and environment. This is not to say that MFC-based controls are not suitable for use in the ActiveX environment, but simply to make you aware of the factors associated with choosing this method. If you are building controls for an Intranet, which is high bandwidth and potentially low latency, the size of the control and the associated DLL are not a major factor. Speed of development, less complexity, and rich features may be more important. In fact, a basic OCX created with the OLE Control Wizard is only 23K. 23K, even on the sluggish Internet, is not extremely large, especially in comparison to some of the large graphic files and AVI files embedded in Web pages. The name of the game is optimization and asynchronous downloading. These topics will be discussed in Chapter 8.

Help is on the way. The Visual C++/MFC team at Microsoft realize that performance is very important in the Internet environment. They are feverishly working to make ActiveX Controls created with Visual C++ and MFC leaner and meaner, as well as working the download of the MFC DLL issue. Visual C++ and MFC may be the best way to create controls, but you will have to weigh each situation accordingly.

MFC Encapsulation of ActiveX and OLE Controls

MFC encapsulates the OLE Control functionality in a Class called COleControl (see Figure 5.17). COleControl is derived from CWnd and in turn from CCmdTarget and CObject.

Figure 5.17. The Class Hierarchy for COleControl.

COleControl is the base class from which you derive to create any OLE Control you want. What's nice is that your control inherits all the functionality of the base class COleControl (see Table 5.5). You can then customize the control to the capabilities you want to include in it. As you know, an OLE Control is nothing more than a COM Object. With MFC, the complexities of dealing with the COM interfaces are abstracted into an easy-to-use class. In addition, MFC provides a framework for your control so you can worry about the details of what you want your control to do instead of recreating functionality that all controls have to contain in order to work.

Table 5.5. The member functions of COleControl that are encapsulated by the Microsoft Foundation Classes.

Function Function COleControl ControlInfoChanged RecreateControlWindow GetClientSite InitializeIIDs GetExtendedControl EnableSimpleFrame LockInPlaceActive SetInitialSize TransformCoords PreModalDialog IsModified PostModalDialog SetModifiedFlag ExchangeExtent ExchangeStockProps GetClientRect OnGetPredefinedValue ExchangeVersion IsConvertingVBX SetModifiedFlag WillAmbientsBeValidDuringLoad DoSuperclassPaint InvalidateControl IsOptimizedDraw SelectFontObject SelectStockFont OnMapPropertyToPage TranslateColor GetNotSupported SetNotPermitted SetNotSupported ThrowError GetReadyState AmbientBackColor InternalSetReadyState AmbientDisplayName Load AmbientForeColor DisplayError AmbientFont DoPropExchange AmbientLocaleID GetClassID AmbientScaleUnits GetMessageString AmbientShowGrabHandles IsSubclassedControl AmbientShowHatching OnClick AmbientTextAlign OnDoVerb AmbientUIDead OnDraw AmbientUserMode OnDrawMetafile GetAmbientProperty OnEdit FireClick OnEnumVerbs FireDblClick OnEventAdvise FireError OnKeyDownEvent FireEvent OnKeyPressEvent FireKeyDown OnKeyUpEvent FireKeyPress OnProperties FireKeyUp OnResetState FireMouseDown OnAppearanceChanged FireMouseMove OnBackColorChanged FireMouseUp OnBorderStyleChanged FireReadyStateChange OnEnabledChanged DoClick OnFontChanged Refresh OnForeColorChanged GetAppearance OnTextChanged SetAppearance OnAmbientPropertyChange GetBackColor OnFreezeEvents SetBackColor OnGetControlInfo GetBorderStyle OnMnemonic SetBorderStyle OnRenderData GetEnabled OnRenderFileData SetEnabled OnRenderGlobalData GetForeColor OnSetClientSite SetForeColor OnSetData GetFont OnSetExtent GetFontTextMetrics OnSetObjectRects GetStockTextMetrics OnGetColorSet InternalGetFont SetFont SelectStockFont GetHwnd GetText InternalGetText SetText OnGetInPlaceMenu GetControlSize OnHideToolBars SetControlSize OnShowToolBars GetRectInContainer OnGetDisplayString SetRectInContainer OnGetPredefinedStrings BoundPropertyChanged BoundPropertyRequestEdit

You are probably wondering why all of these member functions are listed here for you. The purpose is to emphasize the amount of work already done for you by the Microsoft Foundation Classes. In the COleControl class, there are 128 member functions, which when you derive your control from COleControl, your control inherits the capability of using those pre-defined functions.

In addition, MFC itself provides a whole range of capability already created for you when you utilize it. It also includes a functionality to do messaging and automated data exchange.

With MFC Version 4.2, Microsoft added some new classes to MFC to facilitate the creation of ActiveX Controls. These new classes add to MFC's impressive range of functionality. These five new classes are listed in Table 5.6.

Table 5.6. The new classes added to Microsoft Foundation Classes version 4.2 to support ActiveX Controls.

Function Definition CMonikerFile When this class is instantiated as an object, it encapsulates a stream of data named by an IMoniker interface object. It allows you to have access and manipulate that data stream pointed to by an IMoniker object. CAsyncMonikerFile Works much the same as a CMonikerFile except it allows asynchronous access to the IStream object pointed to by the IMoniker object. CDataPathProperty This class encapsulates the implementation of OLE Control Properties so they can be implemented asynchronously. COleCmdUI This class encapsulates the process by which MFC updates the User Interface. COleSafeArray This class encapsulates the function of an array of arbitrary

type and size.

In addition to the new classes in MFC, Microsoft also enhanced COleControl to simplify the creation of ActiveX Controls. These functions add to the already impressive armada of capabilities encapsulated in COleControl. Table 5.7 lists the 31 new member functions added to COleControl.

Table 5.7. The member functions of COleControl that have been added to support ActiveX Controls.

Function Function ClientToParent GetWindowlessDropTarget GetCapture ClipCaretRect GetControlFlags GetClientOffset GetDC GetClientRect InvalidateRgn GetFocus GetActivationPolicy OnGetNaturalExtent OnGetViewExtent ReleaseCapture OnInactiveMouseMove ResetVersion OnQueryHitRect SerializeStockProps SetFocus OnGetViewRect OnGetViewStatus OnInactiveSetCursor OnQueryHitPoint OnWindowlessMessage ReleaseDC ParentToClient ResetStockProps ScrollWindow SerializeVersion SerializeExtent SetCapture

The OLE Control Wizard

The beauty of Visual C++ and MFC is that they perform the mundane task of creating the framework for your control, leaving you the task of making your control perform the functionality you want it to create. At the center of this is the AppWizard, which houses the OLE Control Wizard. Visual C++ 4.2 with the 4.2b patch has augmented the OLE Control Wizard to specifically support ActiveX Controls. In this section, you examine each feature of the Control Wizard and create your first ActiveX MFC Control.

You will need to first launch Visual C++. Once you have Visual C++ up and running, select File from the menu and then New from the pop-up menu. You will then see the New dialog box as shown in Figure 5.18.

Figure 5.18. The New dialog box in Visual C++.

Select Project Workspace. This yields the New Project Workspace dialog box (see Figure 5.19). At the New Project dialog box, you need to select the OLE ControlWizard from the list box on the left, and you need to give your control a title and a location. In this case, call it "Simple Control" and accept the default location.

Figure 5.19. The New Project Information dialog box in Visual C++.

You are now looking at the first page of the OLE Control Wizard (see Figure 5.20). Here the OLE Control Wizard asks you a series of questions about what you would like in your control:

- How many controls do you want in the project?
- Do you want a runtime license for your controls?
- Would you like the Wizard to document your controls with source file comments?
- Would you like a help file generated for your control?

Figure 5.20. Step 1 of the OLE ControlWizard in Visual C++.

In this case, you are going to create only one control, so you select one control for this project. As you have already learned earlier in this chapter, one OCX can contain several controls.

You also select the choice for the Control Wizard to include licensing support for this control. In addition, you ask the Control Wizard to document the code it is going to write for you in the control framework with comments.

Lastly, you ask the ControlWizard to generate a basic help file, so you can provide online help for the Web Masters and programmers who will be utilizing this control. It is extremely important that this control be well documented. You then select the Next button and go to page 2 of the OLE Control Wizard (see Figure 5.21).

Figure 5.21. Step 2 of the OLE ControlWizard in Visual C++.

Step 2 of the OLE ControlWizard presents you with more options for this OLE control.

- Editing the names of the classes for your control.
- What features do you want in your control?
- Activates when visible?
- Invisible at runtime?
- Available in Insert Dialog dialog box?
- Has an About box?
- Acts as a simple frame control?
- Would you like the Wizard to create your control as a subclass to an existing control?
- Would you like advanced ActiveX Enhancements for your control?

The OLE ControlWizard enables you to control the naming of each of the controls in your project (see Figure 5.21) to include the class names, source file names, and property sheet names. If you press the Edit Names button (see Figure 5.21) you will get the Edit Names Dialog as seen in Figure 5.22. It does provide a default naming convention, and in this case, you will accept the defaults provided by the ControlWizard.

Figure 5.22. The Edit Names dialog box in Step 2 of the OLE ControlWizard in Visual C++.

Next are questions regarding what features you want to have in this control. You need to keep in mind the previously discussed section on optimizations. Does the control need to be active when visible, or is it invisible at runtime like a timer control or a communications control? This control will need to be active and visible. You want this control to be available in the Insert Object dialog, so you will choose this option. No doubt you are proud of the controls you create, so you can include an About dialog box to post your name or your company's name. Lastly, do you want this control to be a simple frame control and support the ISimpleFrameSite interface? This is so the control can act as a frame for other controls. For this example's purposes, you will not choose this option.

You now need to take a look and select the advanced options that support ActiveX Enhancements. Click the Advanced button and go to the Advanced ActiveX Controls Features dialog as depicted in Figure 5.23.

Figure 5.23. The Advanced dialog box of Step 2 of the OLE ControlWizard in Visual C++.

From the Advanced ActiveX Controls Features dialog, you can choose one of six options. Keep in mind the previous information you have covered on these options.

- Windowless activation
- Unclipped device context
- Flicker-free activation
- Mouse pointer notification when inactive
- Optimized drawing code
- Loads properties asynchronously

Choose all but Windowless Activation, and click the OK button. Then you need to select the Finish button. Here you will get a summary of the features the OLE Control Wizard will create for you in the New Project Information dialog (see Figure 5.24).

Figure 5.24. The New Project Information dialog of the OLE ControlWizard in Visual C++.

When you click the OK button of the New Project Information dialog, the OLE ControlWizard will create a basic control for you and implement all the features you selected in it. This control need only be compiled and it is up and running. The OLE ControlWizard even added an ellipse in this control's drawing code so it will have something to display. You

now have the framework to start customizing this control. The nice thing is that most of that functionality is already encapsulated in MFC. To assist you in this endeavor, Visual C++ provides you the Class Wizard. The sky is the limit on what types of creations are possible now that you have the framework built for you.

ActiveX Template Library (ATL)

Because MFC-based Controls come with the overhead of the MFC runtime dynamic link libraries, the Visual C++ Development team created the ActiveX Template Library. The ActiveX Template Library is a set of template-based C++ classes to create small fast COM objects. These classes eliminate the need for any external DLLs or any C runtime library code.

In fact, the ATL will produce an in-process server that is less than 5K. Compared to the 22K control plus the 1.4M MFC DLL, that is a significant decrease in size. However, this reduction in size comes with an increased complexity and an increase in the required work to create an ActiveX Control. The ATL does provide all the COM connections for you and a Visual C++ Wizard called the ATL COM AppWizard to guide you in setting up the framework for your control.

The ATL not only allows you to build controls, but also has support for you to build the following COM objects:

- In-process servers
- Local servers
- Service servers
- Remote servers that use the Distributed Component Object Model or Remote Automation
- COM thread models including Single threading, Apartment-model threading, and Free threading
- Aggregatable servers
- Various interface types including custom COM interfaces, dual interfaces, and IDispatch interfaces
- Enumerations
- Connection points
- OLE error mechanisms

Because the ActiveX Template Library provides C++ templates, you have a lot of flexibility to customize a COM object, and in this case this is an OLE Control. As such you utilize the classes by instantiating an instance of the provided class from the template and use it as the basis for your class. This differs from the traditional method of deriving your control's classes from the classes in MFC. This is the distinction between a class library and a template library.

The code in the ATL is highly optimized for the task of creating light, fast COM objects. It still has a lot of the flexibility of the MFC way, and like the MFC way of creating the controls, it keeps you from having to writing a lot of low-level COM code. It requires a through understanding of OLE, COM, and their interfaces.

How to Obtain the ATL

In order to use the ActiveX Template Library, you will first have to download it.

Note

The most current ActiveX Template Library is available from Microsoft, at no charge, at the following Internet URL: http://www.microsoft.com/visualc/v42/atl/default.htm.

The ActiveX Template Library file is a self-extracting archive. In addition, it may also be obtained in future releases of Visual C++. If you subscribe to Level II or higher of the Microsoft Developers Network Library, the ActiveX Template Library should be included in the future releases of MSDN. The location and availability is subject to change by Microsoft.

You will also need Visual C++ 4.1 or greater and the ActiveX SDK. See the previous section's instructions for obtaining the ActiveX SDK and the required Visual C++ components. You must also be running Windows NT 4.0 or greater (non-beta), or Windows 95.

Installing the ATL

The ActiveX Template Library comes in three files: atlinst.exe, docsinst.exe, and sampleinst.exe. The atlinst.exe file contains the ActiveX Template Library. The docsinst.exe contains the documentation for the ATL including setup instructions and a very good white paper. Lastly, the sampleinst.exe file contains some sample applications to guide you in your creation of applications with the ATL.

Caution

The ActiveX Template Library file atltn001.txt file gives instructions for extracting the files through PKUnZip and the -d option. Ignore these instructions because they are incorrect. The files for the ATL are self-extracting and self-installing.

The ATL COM AppWizard

One of the nice features the developers of the ActiveX Template Library included is a Visual C++ Wizard to perform some of the more mundane tasks of creating a framework for a COM object, such as a control. This leaves you the task of making your control perform the functionality you want it to have as opposed to recreating abilities all ActiveX Controls need. The ATL COM AppWizard is the mechanism to create that framework. It will get you up and running quickly, though not as quickly as OLE ControlWizard and MFC.

You will need to first launch Visual C++. Once you have Visual C++ up and running, select File from the menu and then New from the pop-up menu. You will then see the New dialog box.

Select Project Workspace. This yields the Project dialog box. At the New Project dialog box, you need to select the ATL COM AppWizard from the list box on the left, and you need to give your control a title and a location. In this case, call it "ATL Simple Control" and accept the default location. Then press the Create button.

You are now looking at the first page of the ATL COM AppWizard (see Figure 5.25). Here the ATL COM AppWizard asks you a series of questions about what kind of COM Object you want to create.

- How many objects do you want in the project?
- How many interfaces per object?
- Allowing the merging of proxy/stub code?
- Do you want support for MFC?
- What type of registry support would you like? Simple? Advanced?
- What kind of server would you like? DLL? EXE? Service?

- What type of interface would you like? Dual? Custom?

Figure 5.25. Step 1 of the ATL COM AppWizard in Visual C++.

At the top of the Wizard is a spin button with an edit control to enter the number of COM objects you want in your project. Don't worry; you can add more later if you are unsure. However, you will have to do this by hand. You know in this case you are creating the framework for one control, so choose one.

Right next to the number of objects spin button is the interfaces per object spin button. This enables you to generate up to three interfaces for each object. Note that you cannot have a different number of interfaces on different objects. If you want this, you will have to implement it by hand.

When marshaling interfaces are required, you will need to select the Allow merging of proxy/stub code check box. This option places the proxy and stub code generated by the MIDL Compiler in the same DLL as the server. Even though the wizard does some of the work for you, note that in order to merge the proxy/stub code into the DLL, the wizard adds the file dlldatax.c to your project. You need to make sure that precompiled headers are turned off for this file, and you will need to add _MERGE_PROXYSTUB to the defines for the project.

Why the Support MFC check box was included is unclear. The main purpose of using the ATL is to get away from the overhead of MFC. You could have just used the OLE Control Wizard with MFC and saved yourself a lot of time and effort in the first place. However, if for some reason you want to utilize the MFC Class Library, check this option. It will give you access to the MFC Class Library functions.

The ATL COM AppWizard asks you the type of registry support you want for your control. There are two options: Simple (Non-extensible) ATL 1.0 and Advanced (Script Based). The Simple option provides the control with basic self-registration abilities. On the other hand, the Advanced option uses a scripting language. This special scripting language enables you to utilize replaceable parameters during control self-registration.

You now need to select the type of server you want the ATL COM Wizard to create. Your options are: an in-process server (DLL), Local (EXE), or a Service (EXE). When creating a service, you are required to use script-based registration. In addition, when creating a service or executable, you are unable to use MFC or allow merging of proxy/stub code.

The last choice on the ATL COM AppWizard is the type of interfaces to create. The ATL COM AppWizard can create either custom interfaces, derived from IUnknown, or it can create dual interfaces derived from IDispatch.

You then select the next button and go to page 2 of the ATL COM AppWizard.

Step 2 of the ATL COM AppWizard presents you with a single option for this ActiveX control. It enables you to edit the class and the COM names of this control.

Once you are satisfied with the names of your classes and your COM objects, click the OK button. Then click the Finish button on page 2 of the ATL COM AppWizard. The ATL COM AppWizard will then show the New Project Information dialog box. This dialog box shows the selections you have made in creating your COM object.

When you click the OK button of the New Project Information dialog, the ATL COM AppWizard will create a basic COM object for you and implement all the features you selected in it. This control need only be compiled and it is up and running. You now have the framework to start customizing this control.

ActiveX Development Kit (BaseCtl Framework)

The ActiveX Software Development Kit provides another way to produce ActiveX Controls. This is by far the most difficult way to create a control. It is provided by Microsoft as a "bare bones" method of creating a control. This is not for the faint of heart and requires extensive knowledge of OLE, COM, and the OLE Control interfaces. Only minimal

functionality is provided in the code base. You will have to hand code all your messaging, which is a best a daunting task. The only reason to use this method is to try to create the lightest and fastest control possible; however, the time and complexity of creating a control with this method may not be worth the performance gains. This method is not recommended unless you absolutely have to use it. The ActiveX Template Library and Microsoft Foundation Class methods are much easier to implement and much more flexible. Creating and testing a control is no easy task; there are a lot of factors involved. Tools are not talent, but why not use the tools available to make your job easier? As the ATL and MFC methods of creating a control improve, the BaseCtl Framework method in the ActiveX SDK will die away. Some die-hard low-level C programmers or assembly language programmers may want to dive into this low-level approach head first, but make sure you are prepared. It would take a whole chapter to even begin addressing this method of creating a control. The next section covers how to get the framework and set it up, but it is up to you to examine the samples that come with the SDK and explore the quagmire that awaits you in using this method. The authors highly discourage you from using this method.

Getting the BaseCtl Framework

To get the BaseCtl Framework, you will first need to get the ActiveX Software Development Kit. An updated version of the ActiveX SDK was released at same time Internet Explorer 3.0 was released. The new ActiveX SDK was updated to include the new technology and features of Internet Explorer 3.0.

Note

The most current ActiveX SDK is available from Microsoft, at no charge, at the following Internet URL: http://www.microsoft.com/intdev/sdk/.

In addition, you will also need to obtain the Win32 Software Development Kit. The ActiveX SDK requires the August 1996 or later version of the Win32 SDK. The Win32 SDK is available in the Microsoft Developers Network Library Subscription Level II. The required Win32 SDK components come with Visual C++. In addition, you will need to have Visual C++ 4.2b or greater to use the Win32 SDK.

Note

The most current Win32 SDK is available from Microsoft through an MSDN Library Professional Subscription Level II. Microsoft Sales can be reached at 1-800-426-9400.

Once you have the Win32 SDK, you will need to follow the instructions included with it. Pay particular attention to the environment variable that must be set. Included in the Win32 SDK is a SETENV.BAT batch file that will set these Win32 SDK environment variables for you.

If you have Visual C++, your Win32 environment is already set for you.

Setting Up the BaseCtl Framework

The BaseCtl Framework comes on the ActiveX SDK. Once the ActiveX SDK installed, the BaseCtl Framework is in the following location C:\INetSdk\Samples\BaseCtl\Framewrk (assuming that you installed it on your C: drive. You will have to compile the BaseCtl Framework libraries before you can create a control, or compile any of the samples that come with the ActiveX SDK.

To compile the BaseCtl Framework the using the Win32 SDK or another compiler, follow the instructions in the ActiveX SDK. To compile them from the Visual C++ Integrated Development Environment (IDE), follow these instructions.

1. 1. Click the Options… menu item in the Tools popup menu.
2. 2. Click the Directories tab in the Options dialog box.
3. a. Add the full path to the installed components.
4. INetSDK\Include to the includes directories.
5. INetSDK\Lib to the library directories.
6. INetSDK\Bin to the executable files directories.
7. b. These paths must be moved to the top of the search paths.
8. 3. Create a project file and make files.
9. a. Click the New menu item on the File popup menu.
10. b. From the New dialog box, select Project Workspace and click OK.
11. c. Type the name of the project in the New Project Workspace dialog.
12. d. Select what type of application you are building. If you are building an ActiveX Control, choose Dynamic Link Library. If you are going to build the BaseCtrl Framework Libraries, choose Static Library.
13. e. Type the path to the sample or library. You can use the Browse button if you are unsure.
14. f. Click the Create button.
15. g. Click the Files into Project… menu item on the Insert popup menu.
16. h. Select all the *.CPP, *.C, *.DEF, and *.RC files in the directory including any ODL files, and click the Add button.
17. h. Click the Settings… menu item on the Build popup menu.
18. i. Select the Link tab.
19. j. Add the libraries needed to compile the application.
20. i. If the sample you are building does not use MFC, you will need to turn off MFC support in the Project Settings dialog.
21. 4. You can now build the project in Visual C++.

Once you have compiled the debug and release versions of the BaseCtrl Framework, you can start creating your control. You may want to use one of the sample controls as a template, but if you were going to do that you might want to just use MFC or the ATL.

Summary

This chapter discussed the ActiveX Control, which is a superset of the OLE Control. OLE Controls are nothing more than COM objects. ActiveX Controls are a leaner and meaner implementation of the OLE Control to facilitate its use on the Internet. ActiveX/OLE Controls enable you to use prepackaged components of functionality to aid you in creating useful applications.

Chapter 6

Creating OLE Control Containers

By Thomas L. Fredell

Note

The SimpleControl used in this chapter is not the same as the one used in Chapter 5. The same name was inadvertently used in both chapters.

Before OLE Custom Controls, or OCXs/ActiveX controls, the major software component container was Visual Basic. Visual Basic used a proprietary means of communicating with its components, which were called VBXs, or Visual Basic Custom Controls. Using a VBX in a C or C++ application basically meant emulating the specific control containment functionality of Visual Basic.

With the advent of the generic standard of OLE Custom Controls, there are no restrictions on the type of applications that can act as control containers. The minimum requirements for a functional control container are that: 1) it can provide COM-style interfaces to a control; 2) it supports the minimum set of control container interfaces expected by an embedded control; and 3) it implements the minimum number of required functions in the aforementioned interfaces.

The complexity of control containers ranges from full development environments like Visual Basic 4, which allow controls to be integrated into the environment through the toolbar and implement design-mode and run-mode versions, to very simple control containers, like dialog boxes, implemented using the Microsoft Foundation Classes. This chapter discusses the characteristics of control containers and implementation of containers using the Microsoft Foundation Classes.

A Primer on Control/Container Interaction

Control and container interaction is based on the same foundation as the interaction of standard OLE document objects and their containers. Several of the interfaces required for control containers, such as IOleInPlaceSite and IOleInPlaceUIWindow, are used for interaction between OLE compound document objects and their containers. The standard interfaces are supplemented by several required interfaces designed specifically for control and container interaction, such as IOleClientSite and IOleControlSite. All of the required interfaces are listed later in this chapter, with detailed information about the purpose of the interface methods. There are also several optional container interfaces; containers may elect to implement or use them if they require the interface functionality.

One of the most important interfaces required for control/container interaction is IDispatch, the OLE automation dispatch interface. The control exposes an IDispatch interface that its container uses to access the control's properties and methods. The container also exposes an IDispatch interface to the control that allows the control to pass events to its container.

Each control embedded in a container is allocated a control site by the container. The container lays out the contained control, manages keyboard interaction, enables the properties of controls to be saved in some persistent format, handles events generated by controls, and exposes ambient properties to controls. Ambient properties allow OLE controls to retrieve information about the control site provided by their container.

Some containers wrap their controls with extended controls. Extended controls allow containers to maintain additional properties and events for controls without changing the mechanism through which they interact with the controls.

The next few sections describe in more detail the issues involved with creating controls, communicating with them, and wrapping them with extended controls.

Creating an OLE Control Instance

The control container can create a control by retrieving the control's IClassFactory interface through the standard OLE function CoGetClassObject(). The container can then use the ClassFactory::CreateInstance() function to create an instance of the OLE control. ClassFactory::CreateInstance() may return the error CLASS_E_NOTLICENSED if the control requires the container to specify control licensing information before it can be created.

The ClassFactory2 interface handles control licensing; it extends the ClassFactory interface with the GetLicInfo(), RequestLicKey(), and CreateInstanceLic() functions.

```
HRESULT ClassFactory2::GetLicInfo(LICINFO* pLicInfo);
```

GetLicInfo() enables the container to retrieve information regarding the licensing capabilities of the control. The function fills a LICINFO structure whose members describe the control's licensing information on the current machine:

```
typedef struct tagLICINFO
    {
    ULONG cbLicInfo;
    BOOL fRuntimeKeyAvail;
    BOOL fLicVerified;
    } LICINFO;
```

The LICINFO.cbLicInfo structure member simply specifies the size in bytes of the LICINFO structure. The LICINFO.fRuntimeKeyAvail member indicates whether or not the control can be instantiated on an unlicensed machine; if it is TRUE, the control can be created using a key obtained from ClassFactory2::RequestLicKey(). LICINFO.fLicVerified indicates whether or not a full machine license for the control exists.

```
HRESULT ClassFactory2::RequestLicKey(DWORD dwReserved, BSTR* pbstrKey);
```

RequestLicKey() allows a control container to create a runtime instance of a control on a machine that doesn't have a full license for the control (see the following example). If RequestLicKey() succeeds, the control's ClassFactory2 implementation will fill the pbstrKey with a string allocated using SysAllocString(). The container can pass the key to CreateInstanceLic(); the container is responsible for freeing the key using SysFreeString() after it is finished with it.

```
HRESULT ClassFactory2::CreateInstanceLic(IUnknown*
    pUnkOuter, IUnknown* pUnkReserved, REFIID riid,
    BSTR bstrKey, void** ppvObject);
```

CreateInstanceLic() uses the key obtained from the control through RequestLicKey() to create an instance of the control (see the following example). The pUnkOuter argument is a pointer to the controlling unknown for the control if the control is being aggregated; a control container that uses extended controls would pass its IUnknown implementation as an argument. pUnkReserved is a currently unused argument—it must be specified as NULL. The riid argument is the type of interface pointer requested by the container; a container may use the IID_IOleObject to retrieve the control's implementation of the IOleObject interface. The bstrKey is the key obtained using the ClassFactory2::RequestLicKey() function. Finally, the ppvObject is a pointer to the interface returned by the control.

Control/Container Communication Using IDispatch

The primary communication link between a container and a control is through the OLE automation IDispatch interface.

IDispatch, the foundation of OLE automation, provides a generic means through which function calls may be passed to an object. For OLE controls, a control-provided IDispatch allows a container to access control properties and methods. Likewise, a container-provided IDispatch enables controls to access container ambient properties and alert the container of events.

Note

Communication between a control and container is pretty complex. The next few sections describe some of details of the communication, but I should forewarn you that the Microsoft Foundation Classes basically handle all of the details for you.

Fundamentals of IDispatch

The primary IDispatch method used for control/container communication is Invoke(). The prototype for Invoke() is:

```
HRESULT IDispatch::Invoke(DISPID dispidMember,
    REFIID riid, LCID lcid, unsigned short wFlags,
    DISPPARAMS FAR* pdispparams, VARIANT FAR* pvarResult,
    EXECPINFO FAR* pexcepinfo, unsigned int FAR* puArgErr);
```

Callers of IDispatch::Invoke() use a Dispatch ID, or DISPID, to identify the OLE automation member that they want to access. The value of wFlags indicates whether a property or method is the target of Invoke(). Parameters to the property or method are passed to Invoke() as an array of variants, and the return value from the call is used to fill the pvarResult parameter.

Using IDispatch to Access Control Properties and Methods

Controls may contain a large number of properties and methods. As you know from the previous chapter, the capabilities of the control are defined in the control's type information, or type library. Control containers get access to the properties and methods using a control's IDispatch interface.

To get or set a property value, or to call a control method, you use the control's IDispatch::Invoke() implementation. You specify the specific property or method using its dispatch ID; you can determine the dispatch IDs statically by using a tool like the OLE viewer to examine the control's type library, or you can use the control's type information at runtime.

Later in the chapter, the examples use a control called, appropriately enough, SimpleControl. SimpleControl really doesn't do anything interesting; it exposes two properties, no methods, and two events. One of the properties is a SimpleName property with a DISPID of 2. SimpleName is a string; setting the value of SimpleName causes the control to fire an OnSimpleNameChange event. The following code can be used to change the SimpleName property:

```
VARIANT varArg;           // New property value
DISPPARAMS dispparams;    // Structure containing property parameter
```

```
DISPID dispidNamed;       // DISPID of named argument (just one here)
EXCEPINFO excepinfo;      // Contains exception information, should one occur
HRESULT hres;             // Result of property set
UINT uiArgErr;            // Index to erroneous parameter
// Setup new property value parameter
VariantInit(&varArg);
V_VT(&varArg) = VT_BSTR;
V_BSTR(&varArg) = SysAllocString(L"A new name");
// ...setup parameter information...
memset(&dispparams, 0, sizeof dispparams);
dispparams.rgvarg = &varArg;
dispparams.cArgs = 1;
dispidNamed = DISPID_PROPERTYPUT;
dispparams.rgdispidNamedArgs = &dispidNamed;
dispparams.cNamedArgs = 1;
// ...and initialize EXECPINFO structure
memset(&excepinfo, 0, sizeof EXCEPINFO);
// Call Invoke to set the property; assumes pDispatch is the
// interface pointer for SimpleControl's IDispatch
hres = pDispatch->Invoke(2, IID_NULL, 0, DISPATCH_PROPERTYPUT, &dispparams,
    NULL, &excepinfo, &uiArgErr);
if (FAILED(hres))
{
    // Handle error
}
else
{
```

```
    // Call was successful

}

VariantClear(&varArg);
```

The level of effort illustrated in the code snippet is representative of what's necessary to manipulate control properties or methods. You will probably notice that most of the work in manipulating a control's properties or methods is just setting up the parameters. The preceding example is a simple one; if you're dealing with named arguments for method calls, it can get more complex. For more information, you can check out the documentation for the IDispatch interface in the Win32 SDK help.

Using IDispatch to Get Control Events

Controls notify their containers of events using their pointer to the container's IDispatch interface. Before a control can do so, the container must connect its IDispatch to the control as an outgoing interface. To connect its IDispatch, the container must find the control's connection point for the interface.

Connection points are maintained by controls that fire events. The interface to the set of control points, or more precisely IConnectionPoint objects, is through the control's implementation of IConnectionPointContainer. The IConnectionPointContainer interface allows the container to enumerate the control's outgoing interfaces, or to find a specific outgoing interface using an interface ID or IID. For events, the container needs the IID of the control's IDispatch interface with the default and source attributes. You can get the IID statically, using the aforementioned OLE Viewer tool, or you can find it at runtime using the control's type information interfaces. Once the container has acquired the correct IConnectionPoint interface, it can attach its event IDispatch using the IConnectionPoint::Advise() method.

When a control fires an event, it passes the parameters through the standard IDispatch array mechanism. The control uses the DISPID in the call to IDispatch::Invoke() to indicate which event occurred.

Container Ambient Properties

Ambient properties are control container properties that are analogous to control properties. A container maintains ambient properties for all of its controls; typically, the container will expose the same values to all controls within a given frame. The OLE standards define a set of ambient properties with associated IDs that should be implemented by a container. The container also has means through which it can indicate to controls that the ambient properties have changed.

The implementation of control container properties is very similar to the implementation of control properties; container ambient properties are provided by an IDispatch interface that can be retrieved from the IOleControlSite interface that hosts the control. Controls get ambient property values in the same manner that containers get control properties; the container returns the value of the property in the return value from the IDispatch call.

The following list contains the standard ambient properties that may be provided by an OLE control container:

BackColor
Type: OLE_COLOR
Desc: This specifies the color for the control's interior.

DisplayName
Type: VT_BSTR
Desc: The control can use this property to retrieve a container-specified name that it should display in error messages.

Font
Type: OLE_FONT
Desc: This specifies the standard font that the control should use. The control can interpret this property in whatever fashion it chooses.

ForeColor
Type: OLE_COLOR
Desc: This specifies the color for text and graphics in a control.

LocaleID
Type: VT_I4
Desc: This property specifies the locale identifier for the container's UI.

MessageReflect
Type: VT_BOOL
Desc: The MessageReflect property indicates whether or not the container reflects Windows messages back to the control.

ScaleUnits
Type: VT_BSTR
Desc: The container specifies the name of its coordinate units using this property.

TextAlign
Type: VT_I2
Desc: The container uses this property to specify the text alignment in the control. Valid values for the property, with their meaning, are 0[en]general alignment, numbers to the right and text to the left; 1[en]left justify text; 2[end]center justify text; 3[en]right justify text; and 4[en]fill justify text.

UserMode
Type: VT_BOOL
Desc: This indicates the current user interaction mode. If it is FALSE, the user interaction mode is "Design Mode"; otherwise, the user interaction mode is viewing or interacting.

UIDead
Type: VT_BOOL
Desc: The control can use this to determine whether or not the container is allowing user input. A situation in which it is very useful for a control to disallow control interaction is during a container's "Debug" mode (assuming, of course, that the container has such a mode).

ShowGrabHandles
Type: VT_BOOL
Desc: The container uses ShowGrabHandles to indicate to controls whether or not they should display grab handles when they are UI active.

ShowHatching
Type: VT_BOOL
Desc: This property allows the container to indicate the desired hatching feedback for UI active controls.

DisplayAsDefaultButton
Type: VT_BOOL
Desc: Button-like controls use this property to determine if they should display themselves using a visual default button indication (like a thick border).

SupportsMnemonics
Type: VT_BOOL
Desc: Containers indicate support for control mnemonics using this property.

AutoClip

Type: VT_BOOL
Desc: This property indicates whether or not the container will perform automatic clipping for the control.

Some Ambient properties, such as UserMode, may apply to all of the controls within a control container. Other ambient properties, such as BackColor, may depend on the window to which the control is connected.

Extended Controls

Extended controls may be implemented by control containers to provide a wrapper for contained controls. Extended controls allow containers to associate whatever additional information or events they may need to maintain controls. Wrapping a control with an extended control simply involves performing standard COM aggregation. When a container creates a control instance, it passes in a controlling unknown that implements IDispatch for extended properties and events. If a request for a control property is received by the extended control, it passes the request through to the control's interface implementation.

The advantage of wrapping controls with extended controls is that the container can treat its properties and events and the control's properties and events without specialized code. Visual Basic is one of the best examples of a container that implements extended controls due to the seamless integration with its user interface.

Visual Basic and Extended Controls

Visual Basic uses extended controls extensively. Each control embedded in a Visual Basic Form has extended properties that store information such as a control's Top and Left coordinates within a form.

Visual Basic does an excellent job of integrating extended properties and events with standard control properties and events. Earlier in this chapter, you were introduced to the SimpleControl OLE control. SimpleControl has only two properties, SimpleName and SimpleName2, and two events, Click and OnSimpleNameChange. However, after you've added a SimpleControl to a VB form, a whole bunch of properties appear for the SimpleControl object. Figure 6.1 illustrates the properties box for the SimpleControl object in Visual Basic. Notice that SimpleControl's properties are integrated with the other Visual Basic extended properties. From the user's perspective, there is no difference between the properties maintained by the control and the properties maintained by Visual Basic.

Figure 6.1. The Visual Basic Property Editor.

Visual Basic also has a tool called the Object Browser. The Object Browser displays the methods and properties exposed by a Visual Basic control or by a referenced type. A user can add referenced types to Visual Basic using type libraries and Visual Basic's Tools|References… menu option. Figure 6.2 illustrates the display from the Object Browser when SimpleControl is added as a control to a Visual Basic project.

Figure 6.2. The Object Browser's view of SimpleControl as a control.

If SimpleControl is added to a Visual Basic project as a reference instead of a control, only the properties and methods that are part of SimpleControl are visible in the Object Browser. Figure 6.3 shows the Object Browser when SimpleControl is loaded as a reference.

Figure 6.3. The Object Browser's view of SimpleControl as a reference.

As indicated previously, Visual Basic also adds events to its controls. SimpleControl has only Click and OnSimpleNameChange events; Figure 6.4 illustrates SimpleControl's events and the events added by Visual Basic.

Figure 6.4. Visual Basic's Code Editor listing SimpleControl's events.

Unlike ambient property guidelines, there is no standard for the set of extended properties or events implemented by a control

container. Visual Basic provides a good model for extended property integration; it includes properties that handle generic issues like storing control layout information (Top, Left, and so on), and it integrates those properties seamlessly with its user interface.

Required Container Interfaces

There are a substantial number of OLE Interfaces that must be implemented by an OLE control container. The following sections describe the interfaces and the expected behavior of the container upon receiving a method call on the interface.

Some of the methods are indicated as optional methods; the optional methods are not essential for the functioning of controls in a control container. Containers can return E_NOTIMPL or S_OK as appropriate for the optional methods.

IOleClientSite

Controls use IOleClientSite to obtain information about their container, including information that can be used to interact with other controls in the container.

```
HRESULT IOleClientSite::GetContainer(LPOLECONTAINER FAR* ppContainer);
```

The control uses this method to acquire a pointer to the container's IOleContainer interface. A control may use the IOleContainer interface to navigate to other controls contained within the control's container.

```
HRESULT IOleClientSite::ShowObject();
```

The control uses this method to ask its container to show it; this ensures that the control and container are visible:

```
HRESULT IOleClientSite::OnShowWindow(BOOL fShow);
```

The control calls this to notify its container when its window is about to become visible or invisible:

```
HRESULT IOleClientSite::SaveObject();(optional)
```

This method is typically used to save the object that is connected to a client site. The embedded object uses SaveObject() to request its container to save it to persistent storage; the object will probably call this method during the call to its IOleObject::Close() method.

```
HRESULT IOleClientSite::GetMoniker(DWORD dwAssign, DWORD dwWhichMoniker,
```

```
[icc]IMoniker **ppmk);(optional)
```

GetMoniker() is used by an embedded object to request a moniker, which is used to support OLE linking, from its container.

```
HRESULT IOleClientSite::RequestNewObjectLayout();(optional)
```

A loaded but inactive control uses RequestNewObjectLayout() to ask its container to allocate more or less space to display the control. In the implementation of RequestNewObjectLayout(), the container can query the control for the desired size by calling the control's GetExtent() method.

IAdviseSink

Control containers only need to implement IAdviseSink if they want to receive notifications of changes to controls that support IDataObject or IViewObject. Containers may also choose to implement this if they support insertion of controls as standard embedded OLE objects.

```
void IAdviseSink::OnDataChange(FORMATETC* pFormatetc, STGMEDIUM* pStgmed);
```

If a control supports IDataObject, it can notify its container of data changes through this method:

```
void IAdviseSink::OnViewChange(DWORD dwAspect, LONG lindex);
```

If a control supports IViewObject, it can call this container method to indicate that the view of the control has changed:

```
void IAdviseSink::OnRename(IMoniker *pmk);
```

This container member will be called when a control's moniker changes if the control supports linking:

```
void IAdviseSink::OnSave();
```

The control uses OnSave() to notify its container that it has been saved:

```
void IAdviseSink::OnClose();
```

Controls use OnClose() to indicate a transition from running to loaded state:

IOleInPlaceSite

This interface is used to manage interaction between a control container and a control's client site. The client site is the display site for the control; consequently, the interface is derived from the IOleWindow interface. IOleInPlaceSite provides methods that may be used to manage the activation and deactivation of a control, retrieve information about the position in the parent window where the control should place its in-place activation window, manage scrolling of the control, the control's undo state, and control borders.

```
HRESULT IOleWindow::GetWindow(HWND* phwind);
```

The control uses this to retrieve the handle to its in-place window:

```
HRESULT IOleWindow::ContextSensitiveHelp(BOOL fEnterMode);
```

A control can notify its container of a request for context-sensitive help by calling this container method:

```
HRESULT IOleInPlaceSite::CanInPlaceActivate();
```

This is used by the control to determine whether or not the container will allow it to active in place:

```
HRESULT IOleInPlaceSite::OnInPlaceActivate();
```

The control calls this to notify its container that it is in the process of becoming in-place active:

```
HRESULT IOleInPlaceSite::OnUIActivate();
```

The control uses this method to notify its container that it is about to be activated in place. In response and if appropriate, the container should remove whatever user interface is part of its own activation. If a different control is being deactivated as this control becomes active, the container should notify the other control of its state change using its UIDeactivate() method.

```
HRESULT IOleInPlaceSite::GetWindowContext(IOleInPlaceFrame** ppFrame,

[icc]IOleInPlaceUIWindow** ppDoc,LPRECT lprcPosRect, LPRECT lprcClipRect,

[icc]LPOLEINPLACEFRAMEINFO lpFrameInfo);
```

A control calls this on activation to retrieve pointers to the IOleInPlaceFrame and IOleInPlaceUIWindow interfaces provided by its container, the position and clipping rectangles for the control, and an OLEINPLACEFRAMEINFO structure listing accelerators supported by a container during an in-place session.

```
HRESULT IOleInPlaceSite::Scroll(SIZE scollExtent);
```

This is called by a control to request its container to scroll. After the container has finished scrolling, it should check if the control's visible rectangle has been affected. If it has been affected, the container should call the control's SetObjectRects(),

on the IOleInPlaceObject interface, to give the control a new clipping rectangle.

```
HRESULT IOleInPlaceSite::OnUIDeactivate(BOOL fUndoable);
```

The control uses this to notify its container that it is deactivating its user interface components; the container should correspondingly reinstall its user interface. The fUndoable flag indicates whether or not the control can undo changes that occurred; to undo the changes, the container can call the control's OleInPlaceObject::ReactivateAndUndo() method.

```
HRESULT IOleInPlaceSite::OnInPlaceDeactivate();
```

Controls call the container's OnInPlaceDeactivate() method to indicate that they are fully deactivated. After a control has called this method, it is no longer possible for the container to undo changes.

```
HRESULT IOleInPlaceSite::DiscardUndoState();
```

This method is used by the control to indicate to its container that there is no longer any undo state; therefore, the container should not call the control's OleInPlaceObject::ReactivateAndUndo() method.

```
HRESULT IOleInPlaceSite::DeactivateAndUndo();
```

The control calls this if the user invokes undo immediately after activating it. In response, the container should call the control's IOleInPlaceObject::UIDeactivate() method to activate itself, remove the control's UI, and undo changes to the control's state.

```
HRESULT IOleInPlaceSite::OnPosRectChange(LPCRECT lprcPosRect);
```

This is called by a control to indicate a size change; the container should call the control's IOleInPlaceObject::SetObjectRects() to inform the control of the new size and position of the in-place window and new clipping rectangle.

IOleControlSite

Control containers implement this interface to communicate with embedded controls. The following methods are part of the IOleControlSite interface.

```
HRESULT IOleControlSite::OnControlInfoChanged(void);
```

The control calls this method to indicate to the container that the control's control information has changed. The control information is stored within the CONTROLINFO structure; the container can retrieve the updated information from the control using the GetControlInfo method on the control's IOleControl interface. The CONTROLINFO structure contains

control keyboard accelerators and keyboard behavior flags.

```
HRESULT IOleControlSite::LockInPlaceActivate(BOOL fLock);
```

The control calls this to determine whether or not it should remain in-place active even if some type of deactivation event occurs:

```
HRESULT IOleControlSite::GetExtendedControl(IDispatch** ppDisp);
```

This is called by the control to obtain the IDispatch pointer to the extended control implemented by the container. Extended controls are used by the control container to maintain additional properties for a control, such as its X and Y location within the control container. Control containers like Visual Basic use extended controls to implement standard control properties like Top, Left, Height, and Width.

```
HRESULT IOleControlSite::TransformCoords(POINTL* pptlHimetric,

[icc]POINTF* pptfContainer, DWORD dwFlags);
```

The control uses this to convert OLE standard HIMETRIC units in a POINTL structure to the units preferred by the container in a POINTF structure. This method may also be used to do the reverse, that is, convert coordinates from control into standard HIMETRIC units.

```
HRESULT IOleControlSite::TranslateAccelerator(LPMSG pMsg, DWORD grfModifiers);
```

UI Active controls use this method to defer keystroke processing to their container. After a control becomes UI Active, the container transfers keystrokes to the control using the TranslateAccelerator() method on the control's IOleInPlaceActiveObject interface.

```
HRESULT IOleControlSite::OnFocus(BOOL fGotFocus);
```

This is called by the control to indicate to the container whether it has gained or lost input focus; the container can manage Default and Cancel button state accordingly:

```
HRESULT IOleControlSite::ShowPropertyFrame();
```

The control uses this method to request the container to display a property page frame for the control. The container can take the opportunity to create a property frame that includes pages for extended control properties; this ensures that both extended and standard control properties are maintained within a single, consistent user interface.

IOleInPlaceFrame

Controls that have associated frame-level tools, like toolbars or menu items, use this container interface to manage container user-interface changes on control activation. Containers can choose to implement optional methods based on their user-interface characteristics. For example, if a container has toolbars, it may choose to implement the toolbar-oriented negotiation functions like GetBorder() and RequestBorderSpace().

```
HRESULT IOleWindow::GetWindow(HWND* phwind);
```

The control uses this to retrieve the handle to its in-place window.

```
HRESULT IOleWindow::ContextSe
```

```
nsitiveHelp(BOOL fEnterMode);(optional)
```

A control can notify its container of a request for context-sensitive help by calling this container method:

```
HRESULT IOleInPlaceUIWindow::GetBorder(LPRECT lprcBorder);(optional)
```

Controls can use this container method to retrieve the outer rectangle, relative to the frame window, where the control can install its toolbar:

```
HRESULT IOleInPlaceUIWindow::RequestBorderSpace(LPCBORDERWIDTHS
```

```
[icc]pborderwidths);(optional)
```

A control calls RequestBorderSpace() with a rectangle indicating the desired space for a toolbar before attempting to install its toolbar UI. If the container accepts the request by returning S_OK, the control can call SetBorderSpace() to ask the container to allocate the requested space.

```
HRESULT IOleInPlaceUIWindow::SetBorderSpace(LPCBORDERWIDTHS
```

```
[icc]pborderwidths);(optional)
```

The container's SetBorderSpace() method is called when the control requests the allocation of space for the control's toolbar.

```
HRESULT IOleInPlaceUIWindow::SetActiveObject(IOleInPlaceActiveObject *pActiveObject,
```

```
[icc]LPCOLESTR pszObjName);
```

Controls call SetActiveObject() to establish a communication link to the container's frame window:

```
HRESULT IOleInPlaceFrame::InsertMenus(HMENU hmenuShared,

[icc]LPOLEMENUGROUPWIDTHS lpMenuWidths);(optional)
```

This method is called by controls to build up a composite menu containing the container's and control's menu items:

```
HRESULT IOleInPlaceFrame::SetMenu(HMENU hmenuShared, HOLEMENU holemenu,



[icc]HWND hwndActiveObject);(optional)
```

SetMenu() is called by controls to request the container to install a composite menu built up by previous calls to InsertMenus():

```
HRESULT IOleInPlaceFrame::RemoveMenus(HMENU hmenuShared);(optional)
```

The control calls this method to allow the container to remove its menu elements from the composite menu:

```
HRESULT IOleInPlaceFrame::SetStatusText(LPCOLESTR pszStatusText);(optional)
```

A control can call SetStatusText() to request the container to display status text from the control in the container's status line:

```
HRESULT IOleInPlaceFrame::EnableModeless(BOOL fEnable);(optional)
```

The control can call EnableModeless(FALSE) to ask its container to disable any modeless dialog boxes that it may be displaying. After the container has done so, the control may display its own modal dialog. When it is finished, the control should call EnableModeless(TRUE) to re-enable the container's modeless dialogs.

```
HRESULT IOleInPlaceFrame::TranslateAccelerator(LPMSG lpmsg, WORD wID);
```

This method is used to translate keystrokes intended for a container's frame window when a control is active in place.:

IOleContainer

Controls can use a container's implementation of IOleContainer to retrieve information about other controls in the container

or to perform object linking functions.

```
RESULT IOleContainer::ParseDisplayName(IBindCtx* pbc, LPOLESTR pszDisplayName,

UNLONG* pchEaten, IMoniker** ppmkOut);(optional)
```

Controls that support linking use this to ask their container to parse a display name and create a moniker. Containers only need to implement this method if they support links to controls or other embedded objects.

```
HRESULT IOleContainer::LockContainer(BOOL fLock);(optional)
```

If a control and its container support linking, the control will call LockContainer(TRUE) to keep the container running until all link clients have been updated. After the clients have been updated, the control should call LockContainer(FALSE) to remove external locks on the container and allow the container to terminate.

```
HRESULT IOleContainer::EnumObjects(DWORD grfFlags, IEnumUnknown** ppenum);
```

A control can use this method to enumerate all of the controls and objects in its container. It's important to note that the enumerator may not actually return all visible controls in the container because some of them may be standard Windows controls.

Optional Container Interfaces

There are a number of container interfaces that are not required but that may be implemented or supported by containers that require their functionality. The following sections describe the interfaces and the expected behavior of the control or container upon receiving a method call on the interface.

ISimpleFrameSite

Containers can choose to implement support for the ISimpleFrameSite interface if they want to support controls that contain other controls. An example of a control that might make use of this interface is a group box that handles certain interaction characteristics of its contained controls. The purpose of this class is to allow controls to filter messages to controls that they contain while allowing them to defer messages for processing by the root control container.

```
HRESULT ISimpleFrameSite::PreMessageFilter(HWND hwnd,

    UINT msg, WPARAM wParam, LPARAM lParam,

    LRESULT* plResult, DWORD* pdwCookie);
```

This method gives a control the opportunity to process a message that is received by a contained control's window before the contained control does any processing:

```
HRESULT ISimpleFrameSite::PostMessageFilter(HWND hwnd,
    UINT msg, WPARAM wParam, LPARAM lParam,
    LRESULT* plResult, DWORD* pdwCookie);
```

A control can use this method to defer message processing to the control's container after the control and its contained control have had an opportunity to process the message:

IPropertyNotifySink

Containers implement the IPropertyNotifySink interface if they want to receive notifications about control property changes. This is useful, for example, if a container maintains its own property-editing user interface. Because this is an outgoing interface, the container must connect this to the control using the connection point mechanism.

```
HRESULT IPropertyNotifySink::OnChanged(DISPID dispid);
```

The control uses this method to notify the container that the property with the dispatch ID dispid has changed:

```
HRESULT IPropertyNotifySink::OnRequestEdit(DISPID dispid);
```

The control uses OnRequestEdit() to notify its container that one of its properties is going to change. The container can respond with S_OK or S_FALSE. The result S_OK allows the control to proceed with the change; S_FALSE indicates that the container won't allow the control to make the change.

IClassFactory2

Support for IClassFactory2, described earlier in this chapter in the section titled "Creating an OLE Control Instance," allows a control container to support runtime licensing. It is implemented by the control, not the control container. If the control implements it, the container may use it as an alternative to IClassFactory to instantiate a control.

Creating an MFC-Based Dialog Box Control Container

The AppWizard, Microsoft Foundation Classes, and the ClassWizard, all of which are integrated pieces of the Microsoft Developer's Studio, make it trivial to create a dialog box that is a very functional OLE control container.

To create a dialog-based control container, perform the following steps from the Microsoft Developer's Studio:

1. 1. Create a new project named DlgContainer using the MFC AppWizard. The application type should be executable, or exe.
2. 2. On the first wizard page, select Dialog Base" and click Next.
3. 3. On the second wizard page, select the OLE Controls option.

4. 4. Click Finish and OK to allow the AppWizard to create your project.
5. 5. Go to the Resources tab in the project workspace, expand the list of dialog resources by double-clicking the Dialog folder, and double-click to edit the dialog resource with the IDD_DLGCONTAINER_DIALOG identifier.
6. 6. Select Insert, Component from the menu bar. The Component Gallery dialog box will appear. Click on the tab marked "OLE Controls." Figure 6.5 shows the Component Gallery dialog.

Figure 6.5. The Component Gallery dialog.

1. 7. Select a control to insert into the project. I've selected the "Simple Control", indicated with a red "thumbs-up" bitmap, that was generated using the MFC OLE Control classes. Click the Insert button to add the OLE control to your project.
2. 8. After you have clicked the Insert button, a dialog box will appear that asks you to confirm the name of the control wrapper class that will automatically be generated. Figure 6.6 illustrates the Confirm Classes dialog. Select OK to accept the "CSimpleControl" class name. The component gallery will generate C++ header and source files for the CSimpleControl class. Select Close to return to the resource editor.

Figure 6.6. The Confirm Classes dialog.

Note

The "Simple Control" that is being used here only has simple BSTR-based properties; if a control had been inserted that supports fonts or pictures, two additional classes would be created—CFont and CPicture classes.

---

1. 9. On the resource editor toolbar, you will notice an additional item, a red "thumb's up" that is identical to the control bitmap in the Component Gallery. Select the "thumb's up" and draw a "Simple Control" on the Dialog box.
2. 10. Select the new control on the Dialog box, and right-click or press Alt+Enter to display the control properties. Figure 6.7 illustrates the resource editor display after showing the properties. The first page includes properties from the Developer's Studio, the second and third pages are directly provided by "Simple Control," and the last page contains all of the control properties presented by the Developer's Studio in list format.

Figure 6.7. The Resource Editor with SimpleControl's properties.

1. 11. Save the modified dialog resource (File|Save from the menu), then press Control+W to display the Control Wizard.
2. 12. Go the "Member Variables" tab in the Control Wizard, select the CDlgContainerDlg class, and then the IDC_SIMPLECONTROLCTRL1 control ID. Click the Add Variable button—a dialog box will appear prompting for the name of the variable. Enter m_simpctrl for the variable name; the variable category should already be control and the variable type should be CSimpleControl.

That's all that it takes to create an MFC project that includes a dialog box that contains an OLE control! It's truly remarkable considering the amount of effort that would be needed to support the same functionality if you implemented the equivalent dialog from scratch.

The Developer's Studio is doing some pretty interesting things here; first, to create the OLE Control page in the Component Gallery, it's scanning the registry to find controls, extracting their toolbar bitmaps, and finally adding them to what looks like a ListView control for user selection.

Second, the Developer's Studio is generating a C++ class that wraps the OLE control. Each of the properties of the control is exposed through Get/Set class methods; the OLE control methods are likewise exposed as class methods. Later in the chapter, you'll learn about some of the magic behind the implementation of the MFC classes that enable them to take standard C++ function calls and convert them in to IDispatch calls that OLE controls understand.

Third, event information from the control is scanned and included in the ClassWizard Message Maps tab. Figure 6.8 shows the ClassWizard with message maps for SimpleControl.

Figure 6.8. The ClassWizard with SimpleControl messages.

To handle the OnSimpleNameChange event fired by the SimpleControl included in the aforementioned example, all you need to do is select the IDC_SIMPLECONTROL1 object ID and the OnSimpleNameChange message, then click the Add Function button. The ClassWizard integrates window message handling and control event handling into the same easy-to-use user interface. That's very impressive when you consider that the method of retrieving standard window-type events is completely different from retrieving Event Sink-based OLE control messages.

Having read the descriptions of the interfaces required for OLE control containers, and the information about the interaction between control and container, you should be very impressed with the Microsoft Developer's Studio and MFC! The Developer's Studio integrates OLE controls directly into the development environment and alleviates literally all of the work required to implement a control container.

Creating a Non-Dialog Resource-Based MFC SDI Control Container

Creating a non-dialog resource-based MFC SDI Control Container is somewhat more complex than creating a dialog-resource-based control container. MFC includes several types of CView-derived classes. Some of them, like CFormView, use dialog resources to lay out controls. Others, like the base CView class, don't use dialog resources for layout. Adding controls to CView classes is more complex than adding controls to resource-based views because: 1) it requires that you create the control explicitly in code; and 2) it doesn't automatically integrate control events into the ClassWizard.

Luckily, creating the control in code is pretty simple. Unfortunately, handling control events involves significantly more manual code editing. The explanations will cover control events after discussing the basic details of creating a SDI control container.

Create the SDI Container Application

To create a SDI control container, perform the following steps from the Microsoft Developer's Studio:

1. 1. Create a new project using the MFC AppWizard for executables; name the project "SDIContainer".
2. 2. In step 1 of the AppWizard, select "Single document" for the type of application. Click Next twice to go to step 3 of the AppWizard.
3. 3. In step 3 of the AppWizard, make sure that the check box next to "OLE controls" is checked. Click Finish to generate the application.
4. 4. Select Insert|Component… from the Main Menu. The Component Gallery dialog box will launch. Select SimpleControl Control from the OLE Controls tab, then click Insert. Click Confirm in the class confirmation dialog, then click Close in the Component Gallery.

After completing these steps, you will notice that the ClassView tab of the project browser contains a new class—CSimpleControl. The Developer's Studio created this class in the same manner that the CSimpleControl class was created in the previous Dialog Box control container example.

Now the manual code editing begins; open SDIContainerView.h in the code editor and make the changes shown in bold in the following code snippet:

```
Excerpt from SDIContainerView.h -

// SDIContainerView.h : interface of the CSDIContainerView class
```

```
//

////////////////////////////////////////////////////////////////////////////

#include "SimpleControl.h"

class CSDIContainerView : public CView

{

...

protected:

    // m_simpctrl is the member that will be used to

    // manipulate the OLE control

    CSimpleControl m_simpctrl;

// Generated message map functions

protected:

    //{{AFX_MSG(CSDIContainerView)

        // NOTE - the ClassWizard will add and remove member functions here.

        //    DO NOT EDIT what you see in these blocks of generated code !

    //}}AFX_MSG

    DECLARE_MESSAGE_MAP()

    DECLARE_EVENTSINK_MAP()

};

...
```

You'll notice that the basic change is adding the SimpleControl.h header file to the view implementation header file, as well as a new member, m_simpctrl, to access and create an instance of the SimpleControl OLE control. You may not recognize the addition of the DECLARE_EVENTSINK_MAP() macro. That macro sets up the event sink for the view object; if you go back to the previous dialog box control container example, you'll notice that this was added without any intervention on your behalf.

Next, use the ClassWizard to create the CSDIContainerView::OnCreate function to handle the WM_CREATE message for the View class. In the body of CSDIContainerView::OnCreate, add code to create the SimpleControl OLE control instance. The required changes are marked in bold in the following code snippet:

```
Excerpt from SDIContainerView.cpp -

////////////////////////////////////////////////////////////////////////////
```

```
// CSDIContainerView message handlers

int CSDIContainerView::OnCreate(LPCREATESTRUCT lpCreateStruct)

{

    if (CView::OnCreate(lpCreateStruct) == -1)

        return -1;


    // TODO: Add your specialized creation code here

    m_simpctrl.Create(NULL, WS_VISIBLE, CRect(10, 10, 100, 100),

        this, ID_SIMPLECTRL);


    return 0;

}
```

The call to CSimpleControl::Create() initializes the OLE control and attaches it to the CView object (or specifically, in this case, the CSDIContainerView object which is derived from CView). One of the parameters to the call, ID_SIMPLECTRL, hasn't been defined yet. ID_SIMPLECTRL is an arbitrary identifier that will be associated with the SimpleControl instance; the identifier will be used when you make additions to the CSDIContainerView's Event Sink map. You're also missing the implementation of the Event Sink map, which you previously declared in the header using DECLARE_EVENTSINK_MAP(). To add the identifier and Event Sink map, make the following additions to SDIContainerView.cpp:

```
Excerpt from SDIContainerView.cpp -

// SDIContainerView.cpp : implementation of the CSDIContainerView class

//

#include "stdafx.h"

#include "SDIContainer.h"

#include "SDIContainerDoc.h"

#include "SDIContainerView.h"

#ifdef _DEBUG

#define new DEBUG_NEW

#undef THIS_FILE

static char THIS_FILE[] = __FILE__;
```

```
#endif

#define ID_SIMPLECTRL    0

/////////////////////////////////////////////////////////////////////////////

// CSDIContainerView

IMPLEMENT_DYNCREATE(CSDIContainerView, CView)

BEGIN_MESSAGE_MAP(CSDIContainerView, CView)

...

END_MESSAGE_MAP()

BEGIN_EVENTSINK_MAP(CSDIContainerView, CView)

END_EVENTSINK_MAP()
```

Now you have a functional SDI OLE control container application that contains a single OLE control. Properties and methods of the control can be accessed using the member functions of the CSimpleControl class; the only major piece of functionality that's missing is the ability to get events from the control. You've already established the foundation for the event dispatch by including the necessary macro declarations in the CSDIContainer header and implementation files.

Add Event Handling

The SimpleControl OLE control only exposes two events—Click and OnSimpleNameChange. In the previous dialog box control container example, events from the SimpleControl control object were listed directly in the ClassWizard. Unfortunately, such is not the case for SDI or MDI MFC applications; you must add the code manually. As previously indicated, you've already added the necessary event sink macros. Now you need to add entries to the event sink map that correspond to the events you want to capture.

Events are added to the event sink using the ON_EVENT() macro. The details of the ON_EVENT() macro are described later in this chapter; for now, the explanation covers only what is necessary to handle the OnSimpleNameChange event.

To use the ON_EVENT() macro, you need to know the DISPID or member id of the event that you want to capture, and the types of parameters passed to the event. You can get that information using the OLE Object View utility (OLE2VW23.EXE), which can be launched from the Tools menu in the Developer's Studio. To get the parameters and id of the event, open the type information for the SimpleControl control by using the Object|View File|View Type|Library… menu option, and select SimpleControl.ocx from the File Open dialog box. Figure 6.9 illustrates the information from Object View.

Figure 6.9. SimpleControl type information in Object View.

The following is the function prototype reported by the Object Viewer for the OnSimpleNameChange event:

```
VOID OnSimpleNameChange(

     String OldName,

     PTR to String NewName
```

```
    )
```

Translated into a C++ prototype, with the function attribute required for MFC message handlers, that is:

```
afx_msg void OnSimpleNameChange(LPCTSTR OldName, BSTR FAR* NewName);
```

The dispatch ID, or DISPID, for the event is 1; it's listed in the Object Viewer as memid = 0x00000001.

Go ahead and add a member function that handles the event to the CSDIContainerView class. You'll call the member function OnSimpleNameChange() in accordance with the aforementioned prototype. The changes to the SDIContainerView.h and SDIContainerView.cpp are listed following:

```
Excerpt from SDIContainerView.h -

...

    DECLARE_MESSAGE_MAP()

    DECLARE_EVENTSINK_MAP()

    afx_msg void OnSimpleNameChange(LPCTSTR OldName, BSTR FAR* NewName);

};

...

Excerpt from SDIContainerView.cpp -

...

afx_msg void CSDIContainerView::OnSimpleNameChange(LPCTSTR OldName,

    BSTR FAR* NewName)

{

    AfxMessageBox("CSDIContainerView::OnSimpleNameChange");

}

...
```

Now you can add the ON_EVENT() entry to the EVENTSINK map. Add the following changes, marked in bold, to SDIContainerView.cpp:

```
...

BEGIN_EVENTSINK_MAP(CSDIContainerView, CView)

    ON_EVENT(CSDIContainerView, ID_SIMPLECTRL, 1, \
```

```
        OnSimpleNameChange, VTS_BSTR VTS_PBSTR)

END_EVENTSINK_MAP()

...
```

The arguments to the ON_EVENT() macro are pretty straightforward; the name of the class that gets the event, the ID of the control that generates the event (this was the ID that was used to create the control in CSDIContainerView::OnCreate()), the dispatch ID of the event, the name of the CSDIContainerView member function that is called when the event occurs, and the arguments to the event function encoded as strings. The reason why they are encoded as strings will be discussed later in the chapter.

Now test your newly implemented event sink. To do so, you need to write some code that causes the OLE control to fire the OnSimpleNameChange event. Go to the resource tab on the project workspace and edit the main menu, IDR_MAINFRAME. Add a separator after Paste on the Edit menu, and then add a menu item called Change SimpleName, with C as the hot key. Change the ID of the option to IDM_CHANGE_SIMPLENAME (you can do this in the Menu Item Properties dialog). Next, add a message handler to CSDIContainerView using the ClassWizard; the default name for the menu message handler (assuming that you correctly chose the Object ID IDM_CHANGE_SIMPLENAME and the COMMAND message) will be OnSimpleNameChange. Change it so that it is OnMenuSimpleNameChange—then click the Edit Code button.

Make sure that your implementation of the message handler for IDM_CHANGE_SIMPLENAME is as follows:

```
void CSDIContainerView::OnMenuChangeSimpleName()

{

    m_simpctrl.SetSimpleName("ANewName");

}
```

Setting the SimpleName property of the SimpleControl instance causes the OnSimpleNameChange() event to fire; MFC ensures that the CSDIContainerView::OnSimpleNameChange() function is called to handle it.

Details of MFC Control Container Implementation

The Microsoft Foundation Classes do some very interesting things to ease the use of controls in a control container. As previously indicated, the Microsoft Developer Studio contains integrated tools that automatically generate classes to wrap the complexity of the control/container interface. Two of the most interesting and enlightening examples of MFC's control container code are evident in the MFC implementation of property get/sets and the implementation of event handling.

MFC Implementation of Property Get/Sets

The Microsoft Foundation Classes use a unique and intricate, but easy-to-use, mechanism to implement property Get/Set methods for controls embedded in a container. The foundation (no pun intended!) of the mechanism is the COleDispatchDriver class. COleDispatchDriver contains a public function named InvokeHelperV that is used to translate standard C++ style function calls into Dispatch type calls.

There are some interesting problems that MFC tackles here—the arguments to the property Get/Set method are on the stack, which is the convention for C++ function calls. The parameters are also standard C/C++ data types, such as longs.

Furthermore, the fact that exposed C++ member functions can be called as standard C++ member functions is remarkable if you consider that the target for the function call is an OLE control, which uses OLE automation as the means for property and method calls.

C++ format function calls are not, of course, what the IDispatch::Invoke() function expects. The convention for calls using IDispatch::Invoke involves passing parameters as variants in a parameter array to a function called using a DISPID or dispatch id.

To demonstrate the MFC answer to the thorny problem of mapping C++ style calls to IDispatch::Invoke() calls, I created a dialog-based MFC application using the App Wizard. I then added a grid control to the main dialog using the component browser; after saving the modified dialog resource, I used ClassWizard to create a member variable to wrap the Grid in the main dialog class. In the process of creating the member variable, the ClassWizard scanned the OLE type information from the control, and generated gridctrl.cpp and gridctrl.h files.

The gridctrl.h Header File

The gridctrl.h header file contains the definition for a CGridCtrl class, derived from CWnd, that implements the properties and methods exposed by the control as C++ class member functions. An excerpt of the class definition from gridctrl.h follows:

```
...
class CGridCtrl : public CWnd
{
protected:
    DECLARE_DYNCREATE(CGridCtrl)
public:
    CLSID const& GetClsid()
    {
        static CLSID const clsid = { 0xa8c3b720, 0xb5a, 0x101b,
            { 0xb2, 0x2e, 0x0, 0xaa, 0x0, 0x37, 0xb2, 0xfc } };
        return clsid;
    }
...
// Attributes
public:
...
```

```
    short GetRows();
    void SetRows(short);
    short GetCols();
    void SetCols(short);
...
// Operations
public:
    void AboutBox();
    long GetRowHeight(short Index);
    void SetRowHeight(short Index, long nNewValue);
    long GetColWidth(short Index);
    void SetColWidth(short Index, long nNewValue);
...
};
```

The class contains the CLSID of the wrapped control, and a function to access the CLSID, as well as function declarations that correspond to properties and methods exposed by the control.

The GetRows and SetRows functions provide functionality that would be provided inside a container like Visual Basic using the standard assignment operator. In Visual Basic, ctrl.Rows = 10 would therefore correspond to the C++ function call ctrl.SetRows(10).

The interesting part of this equation—the implementation of the property Get/Set functions—is in the automatically generated gridctrl.cpp file.

The gridctrl.cpp Implementation File

The following excerpt from the gridctrl.cpp file illustrates the implementation of some of the aforementioned property Get/Set methods and general method calls:

```
// CGridCtrl
...
/////////////////////////////////////////////////////////////////////////////
// CGridCtrl properties
```

```
...
short CGridCtrl::GetRows()
{
    short result;
    GetProperty(0x8, VT_I2, (void*)&result);
    return result;
}
void CGridCtrl::SetRows(short propVal)
{
    SetProperty(0x8, VT_I2, propVal);
}
short CGridCtrl::GetCols()
{
    short result;
    GetProperty(0x9, VT_I2, (void*)&result);
    return result;
}
void CGridCtrl::SetCols(short propVal)
{
    SetProperty(0x9, VT_I2, propVal);
}
...
////////////////////////////////////////////////////////////////////////////
// CGridCtrl operations
...
long CGridCtrl::GetRowHeight(short Index)
{
```

```
    long result;
    static BYTE parms[] =
        VTS_I2;
    InvokeHelper(0x1f, DISPATCH_PROPERTYGET, VT_I4, (void*)&result, parms,
        Index);
    return result;
}
void CGridCtrl::SetRowHeight(short Index, long nNewValue)
{
    static BYTE parms[] =
        VTS_I2 VTS_I4;
    InvokeHelper(0x1f, DISPATCH_PROPERTYPUT, VT_EMPTY, NULL, parms,
         Index, nNewValue);
}
long CGridCtrl::GetColWidth(short Index)
{
    long result;
    static BYTE parms[] =
        VTS_I2;
    InvokeHelper(0x20, DISPATCH_PROPERTYGET, VT_I4, (void*)&result, parms,
        Index);
    return result;
}
void CGridCtrl::SetColWidth(short Index, long nNewValue)
{
    static BYTE parms[] =
        VTS_I2 VTS_I4;
```

```
    InvokeHelper(0x20, DISPATCH_PROPERTYPUT, VT_EMPTY, NULL, parms,
         Index, nNewValue);
}
...
```

The manifest constants in the parms[] arrays are expanded to strings during precompilation. The output from the precompiler for CGridCtrl::GetRowHeight follows:

```
long CGridCtrl::GetRowHeight(short Index)
{
    long result;
    static BYTE parms[] =
        "\x02";
    InvokeHelper(0x1f, 0x2, VT_I4, (void*)&result, parms,
        Index);
    return result;
}
```

The precompiled output is interesting because it makes the definition of the parms[] array somewhat more clear. It's also interesting because it shows how the mapping from C++ function name to the DISPID required by IDispatch::Invoke occurs; the ClassWizard fills the first parameter to InvokeHelper with the DISPID, in this case the constant 0x1f, of the corresponding GetRowHeight IDispatch-exposed method.

From the definition of the Get/SetRowHeight functions, it's clear that MFC is using the InvokeHelper function to somehow translate the C++ function calls to IDispatch::Invoke() calls.

The InvokeHelper Function

The CGridCtrl's reference to the InvokeHelper() function is a reference to the CWnd::InvokeHelper() function. In turn, CWnd::InvokeHelper() is little more than a wrapper for the COleControlSite::InvokeHelper() function, which is pretty trivial; it sets up a variable argument list and forwards the function call to the COleControlSite::InvokeHelperV() function. Unsurprisingly, the COleControlSite::InvokeHelperV() call still isn't the end of the road; MFC aficionados will attest to the fact that the path of execution through MFC code can be very complex. COleControlSite::InvokeHelperV() ensures that an IDispatch interface pointer has been retrieved for the control object, wraps the pointer using an instance of COleDispatchDriver, and then, finally, calls COleDispatchDriver::InvokeHelperV() to actually make the IDispatch call.

The implementation of the COleDispatchDriver::InvokeHelperV() is very interesting. Each element of the variable argument list, prepared by InvokeHelper(), is converted to a variant and inserted in an instance of the DISPPARAMS structure, which is expected by the standard IDispatch::Invoke() method. During the iteration through the variable argument list, the aforementioned parms[] array is used to set the correct type of the variant and to calculate the correct amount to increment the list pointer using the va_arg() macro.

After the parameter conversion is complete, IDispatch::Invoke() is called using the IDispatch control interface pointer maintained by the COleDispatchDriver class. When Invoke() returns, MFC deallocates temporary memory used for the DISPARAMS structure, checks the return code for the Invoke call, and, if an error occurred during the call, throws a COleDispatchException containing all of the available information from the OLE exception structure.

It should be obvious at this point that the amount of work that MFC shields the casual control container from is just amazing! If MFC weren't used to implement the container, each property call would need to be implemented using the standard IDispatch mechanism, which is tedious at best from C++.

The MFC Implementation of Control Event Handling

Speaking from the perspective of the Microsoft Foundation Classes, control event handling is more or less the opposite of using a control's properties and methods. To use a control's properties and methods, MFC does the work to translate a C++ function call into a call to IDispatch::Invoke(). When handling events, MFC has to take a IDispatch::Invoke() call from a control connected to a container's IDispatch event sink, and convert it into a call to a container's C++ member function.

The Framework for Event Handlers

Event handlers are declared within a CCmdTarget-derived class using the DECLARE_EVENTSINK_MAP() macro. DECLARE_EVENTSINK_MAP() establishes the existence of a table that contains information that MFC requires to dispatch events. It must be supplemented by BEGIN_EVENTSINK_MAP(), ON_EVENT(), and END_EVENTSINK_MAP() macros in the class implementation file.

Earlier, you saw an example where I added the code necessary to instantiate an OLE control in a non-resource-based MFC view class. After adding code to create the control, I added event handling to handle the SimpleControl's OnSimpleNameChange event. Recall the following code snippet:

```
BEGIN_EVENTSINK_MAP(CSDIContainerView, CView)

    ON_EVENT(CSDIContainerView, ID_SIMPLECTRL, 1, \

        OnSimpleNameChange, VTS_BSTR VTS_PBSTR)

END_EVENTSINK_MAP()
```

ON_EVENT() is the macro that establishes an entry in a table of event handlers. When dispatching events, MFC will scan this table to find a function that matches an incoming event. The definition of ON_EVENT(), from AfxDisp.h, follows:

```
#define ON_EVENT(theClass, id, dispid, pfnHandler, vtsParams) \

    { _T(""), dispid, vtsParams, VT_BOOL, \

        (AFX_PMSG)(void (theClass::*)(void))pfnHandler, (AFX_PMSG)0, 0, \

        afxDispCustom, id, (UINT)-1 }, \
```

It contains all of the information that is necessary for MFC to map an IDispatch::Invoke() call to a C++ class function call. As with the aforementioned InvokeHelper() function, used for property set/get calls, the event parameters are encoded as a

string within ON_EVENT() macro use.

MFC Event "Forwarding"

When MFC receives an event from a contained control, MFC attempts to find an entry in the aforementioned event sink map. If MFC finds an entry that matches the event dispatch ID, it checks the parameters in the string parameter signature recorded with the entry. After doing so, it performs an intricate conversion of a DISPPARAM array from an array to arguments on the stack, suitable to be passed to a C++ member function.

The MFC "Event Forwarding" mechanism is one of the most useful elements of the class library; like the MFC property implementation, it shelters the container implementer from the details of managing parameters from IDispatch by converting calls to standard C++ function invocations.

Future Directions with OLE Control Containers

Microsoft's ActiveX internet strategy plays heavily on existing technology such as COM and OLE or ActiveX controls. As you learned from the previous chapter, OLE controls require a large number of interfaces to be implemented. Lots of implementation generally means large controls, and large controls are a problem when users have low bandwidth modem connections. Part of the goal of ActiveX is to simplify the requirements for controls; as the requirements change, it's quite conceivable that the requirements for control containers will also change.

I have no doubt that Microsoft will continually update the Microsoft Foundation Classes to reflect the changing standards for OLE or ActiveX controls. When implementing a control container in the future, it's a safe bet to say that you will probably want to use the Microsoft Foundation Classes, unless you have specific requirements that aren't met by MFC.

Summary

This chapter has presented some of the issues involved with the creation of OLE control containers. OLE controls use literally all of the various OLE technologies; consequently, they are very complex to implement. As you can tell from the number of required interfaces for OLE control containers, it is also very complex to implement a control container.

The Microsoft Foundation Classes radically decrease the amount of code necessary to generate a control container. As a matter of fact, implementing a dialog-resource-based control container is trivial! It's basically a point-and-click affair; as the developer, you have to worry about the logic, not the infrastructure, when you use MFC for control containers.

Chapter 7

Microsoft Internet Explorer 3.0 and Its Scripting Object Model

by Weiying Chen

Internet Explorer(IE) 3.0 has an innovative architecture. Its major component (WebBrowser control) is an ActiveX control, which can be used in any container application for Internet browsing. In addition, the Internet Explorer application is an automation server, which can be used from within any container application to create an instance of Internet Explorer 3.0. IE 3.0 provides a scripting object model accessed through scripting languages such as VBScript and JavaScript. IE 3.0 also provides an ActiveX scripting interface to enable third-party vendors to write their own scripting engines to work inside IE 3.0.

This chapter discusses the IE 3.0 components, ActiveX scripting, the scripting object model, and the ability to add Internet browsing to any application.

IE 3.0 Components

The first major component in IE 3.0 (iexplore.exe) is a shdocvw.dll. This DLL is also called the Microsoft WebBrowser control. The WebBrowser control uses the HTML viewer and Hyperlink object.

The WebBrowser control is an ActiveX control, which can be used inside any control container application, such as Visual Basic, Access, or Visual C++.

This control is a document object container. It provides a single frame in which the user can view and edit all types of ActiveX document objects. It is also a Scripting host, hosting the Scripting engine, such as VBScript and JavaScript.

The frame, the second major component, is developed specifically to house the WebBrowser control. This is the executable that users perceive as the standalone IE 3.0 product.

HTML Viewer

HTML viewer an in-proc server. It is a Document object and can be used in any DocObject container application, such as IE 3.0.

Figure 7.1 shows the HTML viewer in IE 3.0.

Figure 7.1. HTML viewer.

Hyperlink

A *hyperlink* acts as a link to an object at another location(target). The location can be within the application itself or in a different application. The user can click on the link and navigate to an object at another location. A

hyperlink is made up of target's location which is identified by a moniker, a displayable name for the target, a string for the location within target, and a string containing additional parameters.

Microsoft defines the OLE hyperlink interface to abstract the hyperlink features. The WebBrowser control implements this interface to support the hyperlink to any document. For more information on hyperlink, refer to Chapter 21, "Hyperlink Navigation."

Scripting Host and Scripting Engine

IE 3.0 is a Scripting host, which creates a Scripting engine and calls on the Scripting engine to run the scripts. The scripts can be written by the scripting language such as VBScript or JavaScript. Some popular examples of Scripting engine are VBScript and JavaScript. Figure 7.2 illustrates the relationship between IE 3.0 and VBScript, and the scrollbar highlighted in the figure is placed on the web page using the <SCRIPT> tag that follows:

```
<script language="vbscript">

dim i,j

for i=1 to 3

    document.write("<TR>")

    for j=1 to 2

        document.write("<TD>")

        document.writeln("<OBJECTid="'Control'" & CStr(j=i*5-1)")

        document.writeln("WIDTH=171 HEIGHT=21 ALIGN=CEBTER")

        document.writeln("CLASSID=")
```

Figure 7.2. IE 3.0 and VBScript.

The page in Figure 7.2 has six ActiveX controls, displayed as a table in the HTML page. The layout is done by VBScript using the write, writeln method exposed by the document object in the IE 3.0 scripting object model.

Scripting Host Role

The Scripting host must invoke the following methods to interact properly with any Scripting engine. The following pseudocode, which includes a little self-explanation, illustrates the fundamental steps necessary on the Scripting host side.

```
//Create a new  VBScript engine

CoCreateInstance(CLSID_VBScript,, . . .)

//Load the script to feed the script engine or create a null script

//import each top-level named entity such as forms and pages into

//the scripting engine's name space

IActiveScript::AddNamedItem()

//Causes the scripting engine to run the script

IActiveScript::SetScriptState(SCRIPTSTATE_CONNECTED)
```

Scripting Engine Role

A Scripting engine must implement a few fundamental steps to communicate with the Scripting host. These steps are demonstrated in the following pseudocode.

```
//Associate a symbol with a top-level item

IActiveScriptSite::GetItemInfo

//Hookup the events with all the relevant objects

IConnectionPoint

//Refers to the OLE object's methods and properties

IDispatch.Invoke
```

Microsoft defines an interface that allows a Scripting engine to be used in the Scripting host. This is known as the ActiveX Scripting interface. Besides the interface required for the Scripting engine, a particular registry key called OLEScript without any values must be present as an immediate subkey for the string representation of the Scripting engine's CLSID.

Scripting Engine Registry Settings

Figure 7.3 illustrates the VBScript ProgID registry setting.

Figure 7.3. VBScript Engine's ProgID registry setting

VBScript in Figure 7.3 is the ProgID for the VBScript engine. The ProgID is used to register the human-readable string associated with the COM object. VBScript engine is a COM object which implements IUnknown interface. The value associated with the ProgID is used in to display the name of the COM object.

There are two immediate subkeys under VBScript ProgID, one is the OLEScript without any values. This subkey is required for any Scripting engine. The other subkey is the CLSID, its value contains the string representation of the Scripting engine's CLSID. The string representation of the CLSID is denoted as {CLSID}.

Figure 7.4 illustrates the registry entry associated with the VBScript engine's {CLSID}.

Figure 7.4. VBScript Engine's {CLSID} registry setting

Figure 7.4 is the registry information stored under {CLSID}. There are two subkeys under "Implemented Categories" root key.

```
{F0B7A1A1-9847-11CF-8F20-00805F2CD064}
```

indicates that VBScript engine is an Active Scripting Engine.

```
{F0B7A1A2-9847-11CF-8F20-00805F2CD064}
```

indicates that the VBScript engine is an Active Scripting Engine with Parsing.

The subkeys under "Implemented Categories" root key are from Component Category. Component Category is a new way to categorize COM objects.

For more information on the ActiveX scripting, please refer to:
http://www.microsoft.com/intdev/sdk/docs/iexplore/.

Scripting Object Model

IE 3.0 Scripting Object Model defines a set of objects which is accessible through any scripting languages such as VBScript and JavaScript. The IE 3.0 object model is compatible with the object model used in JavaScript in Netscape Navigator. ActiveX Scripting defines the interface for the scripting engine and scripting host, whereas the Scripting Object model defines a set of objects accessible by any scripting language.

Figure 7.5 shows the IE 3.0 scripting object model hierarchy.

Figure 7.5. IE 3.0 scripting object model.

In this scripting object model shown in Figure 7.5, window is at the top of the object model. It consists of document, history, and other properties, methods, and events. The document consists of form, link, anchor, write, and so forth. The form consists of action, submit, and elements.

The elements within the form object refer to the HTML intrinsic controls or objects inserted in HTML through the <object> tag. The HTML intrinsic control is built in the Web browser and placed in the <input> tag. Figure

7.6 illustrates the elements, its methods, properties, and events.

Figure 7.6. HTML intrinsic controls.

The following section uses VBScript and JavaScript to demonstrate how to use some of the objects, its events, properties, and methods.

Window

The window object refers to the Internet Explorer window. The properties and methods can be invoked by the script directly. The following example demonstrates a perfectly legal way of calling alert now() without referencing the window object. Normally, you will use window.alert now().

```
<script language="vbscript">

    sub window_onload

        alert now()

    end sub

</script>
```

Tables 7.1 through 7.3 list the definitions of the window object's properties, events, and methods.

Table 7.1. Window object's properties.

Property Meaning get or set name Name of the current window get parent Window object of the window's parent get self Window object get top Window object of the topmost window get location Window's location object get defaultStatus Default status text in the browser's status bar get and set status Status of the browser's status bar get and set frames The array of frames of the current window get history History object of the current window get navigator Navigator object of the current window get document Document object of the current window get

Table 7.2. Window object's methods.

Method Meaning Return Usage
Value alert Displays an alert message box none alert "hello" confirm Displays a message box true/false confirm with OK or cancel selection "continue" prompt Prompts the user for input string prompt "age","28" open Creates a new window object open "http://..." close Closes the window string close navigate Navigates the window to long navigate "http://.."
a new URL setTimeOut Sets a timer to call a long setTimeOut("button1.click",
function after a specified 1000)
number of milliseconds clearTimeOut Clear the timer with ID none clearTimeOut ID
returned by the setTimeOut

Table 7.3. Window object's events.

Event Meaning Parameter Usage OnLoad Fires when the page is loaded none OnLoad="foo" OnUnLoad Fires when the page is unloaded none OnUnLoad="foo"

The following example demonstrates how to change the browser's status by changing the window's status property.

```
<html>

<form name="statusform">

     enter information that will show in the browser's status bar

    <input type=text name=txtStatus>

    <input type=button name=btnChange


[icc]value="click to display on the status bar">

</form>

<script language="vbscript">

    sub btnChange_onClick

        status = document.statusForm.txtStatus.value

    end sub

</script>

</html>
```

As far as window's onload event, the initialization code should be placed in this event. The cleanup code on exiting the window should be placed in the OnUnLoad event.

History

The History object exposes the properties and methods on the current history list. Tables 7.4 and 7.5 enumerate its properties and methods.

Table 7.4. History object's property.

Property Meaning get or set length Length of the history list get

Table 7.5. History object's method.

Method Meaning Return Value Usage back Jumps back *n* steps in the history list none back 3 forward Jumps forward *n* steps in the history list none forward 3 go Goes to the *n*th item in the history list none go 3

There are no events fired by the history object.

Navigator

The Navigator object returns the browser's information. Table 7.6 lists all the properties exposed by the Navigator object. These properties can be accessed in the scripting language by prefixing navigator in front of these properties.

Table 7.6. Navigator object's properties.

Property Meaning get or set appCodeName Browser's code name get appName Browser's name get appVersion Browser's version get userAgent Browser's user agent get

There are no methods exposed, and events fired by the navigator objects, s

Document

The Document object refers to the HTML document in the browser. It can be called by prefixing document in front of the properties and methods in the script. Tables 7.7 and 7.8 enumerate the exposed properties and methods.

Table 7.7. Document object's properties.

Property Meaning get or set linkColor Current color of the link get and set alinkColor Current color of the active link get and set vlinkColor Current color of the visited link get and set bgColor Current color of the background get and set fgColor Current color of the foreground get and set location Location object get lastModified Date which the document was last modified get title Document title get cookie Cookie for the current document get and set referrer URL of the referring document get anchors Array of anchors in a document get links Array of links in a document get forms Array of forms in a document get

Table 7.8. Document object's methods.

Method Meaning Return Usage
Value write Places the string into the current document none write string writeln Places the string into the current document with a newline character none writeln string open Opens the document for output none open [MIME type] close Closes the document and writes the data to the screen none close clear Clears the content of the document none clear

There are no events fired by the document object.

Among these document properties, cookie is very useful to pass information between HTML pages. There is an example at http://www.microsoft.com/vbscript/us/vbssamp/cookies/extcookie.htm showing you how to use cookies.

The following example uses referrer to tell the user where they came from (the original page).

```
<html>

<body onLoad="FindWhereFrom()">

<script language="Javascript">

function FindWhereFrom()

{

    var foo = document.referrer

    if ( foo="http://www.microsoft.com/")

    {

        alert('you came from microsoft home page')

    }

}

</script>

</body>

</html>
```

The methods write and writeln are useful for generating dynamic Web pages. The following example dynamically constructs an HTML table with an array of controls by using write and writeln methods. Figure 7.2 shows this HTML page.

```
<html>

<table>

<script language ="vbscript">

dim i, j

for i=1 to 3

    document.write("<TR>")

    for j=1 to 2

        document.write("<TD>")

        document.writeln("<OBJECT id= ""Control""& CStr(j+i*5-1)")

        document.writeln("WIDTH=171 HEIGHT=21 ALIGN=CENTER")

        document.writeln(" CODEBASE= ""http://")

        document.writeln(" activex.microsoft.com/controls/mspert10.cab""")

        document.writeln(" CLASSID=")

        document.writeln("""clsid:DFD181E0-5E2F-11CE-A449-00AA004A803D "">")

        document.writeln("<PARAM NAME=""Size"" VALUE=""4516;564"">")

        document.writeln(" <PARAM NAME=""Max"" VALUE=""255"">")

        document.writeln("<PARAM NAME=""LargeChange"" VALUE=""51"">")

        document.writeln("<PARAM NAME=""Orientation"" VALUE=""1"">")

        document.writeln("</OBJECT>")

        document.write("<TD>")

    next

    document.write("</TR>")

 next

</script>

</table>
```

```
</html>
```

The location object represents the current URL. The properties exposed by the location object can be accessed within the scripting language by prefixing location in front of the properties. Modifying the properties of this object will force the browser to navigate to the newly constructed URL.

Table 7.9 enumerates the exposed properties.

Table 7.9. Location object's properties.

Property Meaning get or set href Location's complete URL get and set protocol URL's protocol portion get and set host URL's host and port portion get and set hostname URL's hostname portion get and set port URL's port portion get and set pathname URL's pathname portion get and set search URL's search portion get and set hash URL's hash portion get and set

Form

The form object refers to a form within the HTML document. The document object keeps track of forms as an array or by name. The form can be accessed either by index or by name. For instance, for the following page:

```
<form name="test">
    <input type=button name=btnChange value="submit">
</form>
<form name="test1">
    <input type=button name=btnChange1 value="submit">
</form>
```

form "test" is the first form in the document. The following syntax can be used to refer to the "test" form via giving index.

```
document.form[0]
```

Or by referring to the form name "test"

```
document.test
```

Tables 7.10, 7.11, and 7.12 enumerate the exposed properties, methods and events of the form object.

Table 7.10. Form object's properties.

Property Meaning get or set action Address of the form's action get and set encoding Encoding for the form get and set method How the form data is sent to the server set target Name of the target window set elements Array of elements in the form get

Table 7.11. Form object's method.

Method Meaning Return Value Usage submit Submit the form none form.submit

Table 7.12. Form object's event.

Event Meaning Parameter Usage onSubmit Fires when the form is submitted none form.OnSubmit=string

The submit method can be used to validate the client side information before sending to the Web server. For instance, the following example in Listing 7.1 demonstrates how to validate the user's input before sending the information to the Web server.

Listing 7.1: Client Side Validation By using Submit method

```
<form name="validform" action="test.dll" method="post" >
    <input name=txtNumber  type=text size="2">
    <input name=btnSend type=button value="submit">
</form>
<script language="vbscript">
sub btnSend_onClick
    dim value
    value = document.validform.txtNumber.value
    if IsNumeric(value) then
        if value  < 1 or value  > 10 then
            msgbox " please enter a number between 1 and 10"
        else
```

```
            document.validform.submit

        end if

    else

        msgbox "please enter a numeric number"

    end if

end sub

</script>
```

In Listing 7.1, document.validform is used across the html page to refer to the same form. The following code will simply the process.

```
dim theForm

set theForm = document.validform
```

After the above definition, theForm can be used to directly refer to the form in the HTML page.

In Listing 7.1, the information will be validated first when the "submit" button is clicked. If the information provided is not in the expected range, less than 1 or greater than 10, message " please enter a number between 1 and 10" will be displayed. If there is not any information provided, message "please enter a numeric number" will be displayed. This greatly improve the performance, and avoid sending information to the server side to do the validation , and sending back to the client etc.

Link

A link object is constructed for every link in the HTML document. It can be accessed in the scripting language through the indexed array by prefixing the document object reference. For instance, the following example will set the Link to the first link on the page:

```
theLink = document.link(0).href
```

Tables 7.13 and 7.14 show the properties exposed and events fired by the link object. There are no methods exposed by the link object.

Table 7.13. Link object's properties.

Property Meaning get or set href Link's complete URL get protocol URL's protocol portion get host URL's host

and port portion get hostname URL's hostname portion get port URL's port portion get pathname URL's pathname portion get search URL's search portion get hash URL's hash portion get target URL's target portion get

Table 7.14. Link object's event.

Event Meaning Usage OnMouseMove Fires when the mouse move over a link OnMouseMove(shift, button, x, y) OnClick Fires when the mouse is clicked on a link OnClick OnMouseOver Fires when the mouse moves over a link OnMouseMove

Anchor

The anchor object is constructed for every anchor tag in the HTML document. It is accessed through the indexed array as the link object.

Table 7.15 shows the property exposed by the anchor object. There are not methods exposed and events fired by the anchor object.

Table 7.15. Anchor object's property.

Property Meaning get or set name Name of the anchor get and set

For more information on the Internet Explorer scripting object model, please refer to http://www.microsoft.com/vbscript.

Adding Internet Browsing to Any Application

The major component of IE 3.0 is the WebBrowser control. The reason that IE 3.0 has an innovative architecture is that the WebBrowser control can be used in any container application to provide the Internet browsing capability. WebBrowser control provides the functionality such as data downloading, hyperlinking and URL navigation, history information, and parsing and displaying HTML-encoded documents.

Building a Custom Web Browser

The following example illustrates how seamlessly the WebBrowser control integrates with container applications. Here, the language is Visual Basic 4.0 Enterprise Edition. The development environment is NT 4.0 and IE 3.0 release version is used.

Figure 7.7 shows how the application looks.

Figure 7.7. Custom Web browsing application.

The code for this project, shown in Listing 7.2, is fairly self-explanatory. The whole VB project for this sample is contained on the CD, called lst91.vbp.

Listing 7.2. Web browser integration.

```
Private Sub btnGo_Click()

    WebBrowser1.Navigate txtAddress

End Sub

Private Sub imageBack_Click()

    'go back one item in the history list

    WebBrowser1.GoBack

End Sub

Private Sub imageForward_Click()

    'go one item forward in the history list

    WebBrowser1.GoForward

End Sub

Private Sub imageHome_Click()

    'go to the current page or start page

    WebBrowser1.GoHome

End Sub

Private Sub imageRefresh_Click()

    'reload the page that is displaying

    WebBrowser1.Refresh

End Sub

Private Sub imageSearch_Click()

    'go to the current search page specified
```

```
    'in the internet control panel and the IE option
    'dialog box
    WebBrowser1.GoSearch
End Sub
Private Sub imageStop_Click()
    'stop the downloading or navigation
    WebBrowser1.Stop
End Sub
Private Sub WebBrowser1_BeforeNavigate(ByVal URL As String,
[icc]       ByVal Flags As Long, ByVal TargetFrameName As String,
[icc]       PostData As Variant, ByVal Headers As String, Cancel As Boolean)
    Debug.Print "beforeNavigation"
End Sub
Private Sub WebBrowser1_CommandStateChange(ByVal Command As Long,
                                           ByVal Enable As Boolean)
    Debug.Print "CommandStateChange"
End Sub
Private Sub WebBrowser1_DownloadBegin()
    Debug.Print "DownloadBegin"
End Sub
Private Sub WebBrowser1_DownloadComplete()
    Debug.Print "DownloadComplete"
End Sub
Private Sub WebBrowser1_FrameBeforeNavigate(ByVal URL As String,
```

```
[icc]               ByVal Flags As Long, ByVal TargetFrameName As String,
[icc]               PostData As Variant, ByVal Headers As String,
[icc]               Cancel As Boolean)
    Debug.Print "FrameBeforeNavigate"
End Sub
Private Sub WebBrowser1_FrameNavigateComplete(ByVal URL As String)
    Debug.Print "FrameNavigateComplete"
End Sub
Private Sub WebBrowser1_FrameNewWindow(ByVal URL As String,
[icc]         ByVal Flags As Long, ByVal TargetFrameName As String,
[icc]         PostData As Variant, ByVal Headers As String,
[icc]         Processed As Boolean)
    Debug.Print "FrameNewWindow"
End Sub
Private Sub WebBrowser1_NavigateComplete(ByVal URL As String)
    Debug.Print "NavigateComplete"
End Sub
Private Sub WebBrowser1_NewWindow(ByVal URL As String,
[icc]            ByVal Flags As Long, ByVal TargetFrameName As String,
[icc]            PostData As Variant, ByVal Headers As String,
[icc]            Processed As Boolean)
    Debug.Print "NewWindow"
End Sub
Private Sub WebBrowser1_ProgressChange(ByVal Progress As Long,
[icc]                                  ByVal ProgressMax As Long)
```

```
    Debug.Print "ProgressChange"
End Sub
Private Sub WebBrowser1_PropertyChange(ByVal szProperty As String)
    Debug.Print "PropertyChange"
End Sub
Private Sub WebBrowser1_Quit(Cancel As Boolean)
    Debug.Print "Quit"
End Sub
Private Sub WebBrowser1_StatusTextChange(ByVal Text As String)
    Debug.Print "StatusTextChange"
End Sub
Private Sub WebBrowser1_TitleChange(ByVal Text As String)
    Debug.Print "TitleChange"
End Sub
```

In order to create any application using this control, make sure to select the Microsoft Internet control checkbox in the Visual Basic 4.0 custom control list. Tables 7.16, 7.17, and 7.18 enumerate the properties, methods exposed, and events fired by the WebBrowser control.

Table 7.16. WebBrowser control's properties.

Property Meaning get or set Application Application that uses the Web browser control get Busy Whether downloading or navigation is going on get Container Control container get Document Active document get Height Browser control height get and set Width Browser control width get and set Left Distance between the internal left edge get and set of the control and the left edge of the container Top Distance between the internal top edge get and set of the control and the top edge of the container Type Type name of the contained document get LocationName Title of the page or the UNC the browser get
is displaying LocationURL URL of the resource that the browser get
is displaying Parent Parent of the browser control get TopLevelContainer Whether the given object is a top-level container get

Table 7.17. WebBrowser control's methods.

Method Meaning Usage GoBack Go one item back in the history list goback GoForward Go one item forward in the history list goforward GoHome Go to the current home or start page gohome GoSearch Go to the current search page (this search page gosearch
is specified by the Internet control panel and
the IE option dialog box) Navigate Navigate to URL or a full path navigate URL Refresh Reloads the page that is displaying refresh Refresh2 Reloads the page that is displaying refresh [0...3]
with specified level: 0: Normal refresh 1: Refresh if the page has expired 3: Full refresh Note: 2 is not defined. Stop Cancel any pending navigation or download stop

Table 7.18. WebBrowser control's events.

Event Meaning BeforeNavigate Fires before navigating to a different URL CommandStateChange Fires when the enabled state of a command changes DownloadBegin Fires when a download starts DownloadComplete Fires when a download successfully ends FrameBeforeNavigate Fires when navigating to a different URL FrameNavigateComplete Fires when a navigation successfully ends FrameNewWindow Fires when a new window is to be created NavigateComplete Fires when successfully navigated to a new location NewWindow Fires when a new window is to be created ProgressChange Fires when the progress of download is updated StatusTextChange Fires when the status bar text is changed TitleChange Fires when the page title is changed

For more detailed information on these properties, methods, and events exposed by this control, please refer to http://www.microsoft.com/intdev/sdk/docs/iexplore/.

InternetExplorer Object

In addition to using the Web browser control inside a container application, IE 3.0 is also built as an OLE automation server. The ProgID for this automation server is InternetExplorer.Application. By calling as follows, an instance of Internet Explorer 3.0 will be created.

```
createobject("InternetExplorer.Application")
```

The methods and properties supported by the InternetExplorer object are a superset of the Web browser control. In other words, the InternetExplorer object exposes all the properties, methods, and events the WebBrowser control has. In addition, it supports more.

For more information about InternetExplorer Object, please refer to http://www.microsoft.com/intdev/sdk/docs/iexplore/.

Summary

The Internet Explorer 3.0 provides an innovative architecture, which allows developers to add Internet browsing to any application. The ActiveX scripting provides the interface so that scripting engines can be used in the IE

3.0. Furthermore, the IE 3.0 scripting object model provides support for programmatically controlling navigation, history, document, and forms directly from scripting languages.

Chapter 8

VBScript

by Jon Czernel

Microsoft has for years pushed BASIC as the de facto, entry-level development standard in the personal computer arena. In fact, the origins of the Microsoft empire can be traced back to its early adoption and development of BASIC for the first personal computers available. Many developers today, in fact, will say that one of the several Microsoft BASIC products were responsible for their entry into the development arena, either by hobby, profession, or both.

VBScript continues Microsoft's support of a language that has evolved into a fairly powerful development tool. Microsoft has announced that it will allow third parties to use VBScript as the scripting engine in their applications, and it has openly encouraged third parties to port VBScript to "alternative" (non-Windows) platforms through purchase of their VBScript source code.

This chapter examines the fundamental language constructs of VBScript, a strict subset of Visual Basic 4.0 (VB) and Visual Basic for Applications (VBA). In particular, the later sections of this chapter concentrate on the use of VBScript in HTML documents in conjunction with ActiveX Controls. For the sake of brevity, the authors assume that the reader has some development experience in at least one structured development language, such as VB, C/C++, or Object Pascal. Instead of concentrating on syntax, the chapter presents real-world VBScript examples wherever appropriate. The authors also assume that the reader has experience in basic HTML development and is familiar with the concept of ActiveX (OLE/OCX) controls.

This chapter is broken down into the following areas:

- It begins with language fundamentals, discussing the core constructs of the language, without the overhead of HTML and ActiveX.
- For those with VB/VBA development experience, the section titled "Comparing VBScript to Visual Basic," is for you. This is a "must-read" section for experienced VB developers.
- The remaining sections detail the use of VBScript in an HTML document, including the use of VBScript to "glue" together various ActiveX controls in the creation of an interactive Web page.

Note

The complete richness of VBScript is impossible to reveal in a single chapter. For more information on VBScript, refer to the Microsoft VBScript Web site at: http://www.microsoft.com/vbscript.

May your adventures in VBScript be both challenging and fulfilling! The first section begins with a discussion of VBScript fundamentals.

Language Fundamentals

This section discusses the fundamental architecture of VBScript. This section should be used as both a quick language reference and brief language tutorial. Like most language references, the section begins with the most basic elements, and then works its way up to larger, more encompassing constructs.

The VBScript Data Type: Variant

Unlike most other languages, VBScript allows only one data type: *variant*. A variant is a variable type that can hold any type of fundamental data type, including integers, floating points, characters, strings, and date/time values. Variant data types may also represent instances of objects.

With variants, you don't have to worry about ensuring that your variable is adequately prepared to handle unexpected data. However, although the variable itself might be able to contain any type of data, your routines will often be required to check for the type of data that is stored in a variable to ensure proper script execution.

VBScript assigns a *subtype* to variables dynamically during program execution. This subtype classification flags the variable as containing a particular type of value, such as an integer or a string. VBScript also includes the function VarType, and functions such as IsArray(), IsDate() and IsEmpty(), that enable you to determine, at runtime, the currently assigned subtype of a variable.

Table 8.1 enumerates the variant subtypes available, and the value returned by the VarType function when a variable of the given subtype is passed as a parameter. Alternative methods of determining a variable subtype, such as IsEmpty() or IsDate(), are also mentioned.

Table 8.1. Common variant subtypes and descriptions.

Variant VarType Explanation
Subtype Empty 0 Used to indicate an uninitialized variable or a zero-length string. See also: IsEmpty(). Null 1 Used when a variable has been assigned a Null value. See also: IsNull(). Boolean 11 Either True (–1) or False (0). Byte 17 An integer from 0 to 255. See also: IsNumeric(). Integer 2 An integer from –32768 to 32767. See also: IsNumeric(). Long Integer 3 An integer from –2,147,483,648 to 2,147,483,647. See also: IsNumeric(). Single 4 A floating-point value from –3.402823E38 to –1.401298E-45, 0, or 1.401298E-45 to 3.402823E38. See also: IsNumeric(). Double 5 A double precision floating-point number from –1.79769313486232E308 to –4.94065645841247E-324, or 4.94065645841247E-324 to 1.79769313486232E308. See also: IsNumeric(). Date/Time 7 A valid date/time value. See also: IsDate(). Date/Time values may be enclosed in quotes, such as "8/30/96" or in pound characters, such as #08/30/96#. String 8 A variable length string. Strings are enclosed in quotes, such as "Howdy". Object 9 An instance of an automation object. See also IsObject(). Object 13 An instance of an non-automation object. See also: IsObject().

Note that a variable subtype is dynamic during code execution, and it may change as code is executing, depending on the contents of the variable. For this reason, VBScript also includes mechanisms that enable you to convert (cast) a variant variable from one subtype to another. Table 8.2 lists some of the most important conversion functions built into VBScript.

Table 8.2. Variant conversion functions.

Function: Returns Asc() The ASCII code for the first character in the given string. CBool() A Boolean. CByte() A Byte. CDate() A Date/Time. CDbl() A Double Precision number. Chr() The character represented by the given ASCII code. CInt() An integer. CLng() A long integer. CSng() A single precision floating-point value. Hex() The hexadecimal representation of the given number, in string format.

Oct() The octal representation of a given number, in string format.

Warning

Remember that although variants provide a great deal of flexibility, it is still possible to raise type mismatch errors at runtime by casting a variable that contains an invalid value for the specified cast operation. For example, the function CDate will produce a type mismatch error if T contains anything besides a valid date/time value, such as "SomeGuyNamedJoe". Before forcing a variable cast, ensure that the variable you'll be casting contains a valid argument by using one of the Is.. functions, such as IsDate().

Variable Naming Rules

Variable names must conform to the following rules:

1. 1. A variable name must begin with an alphabetic character. The remainder of the name may contain any alphanumeric characters, including underscores ("_").
2. 2. The length of a variable cannot exceed 255 characters.
3. 3. Periods may not be embedded in a variable name.
4. 4. A variable name must be unique within the scope in which it is defined.

Variable Declarations

Variables in VBScript are declared *explicitly* using the Dim statement. Variables that are not explicitly declared are *implicitly* declared when they are first encountered in any statement.

Tip

It is best to explicitly declare your variables before they are used for the first time, as opposed to allowing VBScript to implicitly declare them when they are first used. Explicit declaration makes your code easier to debug and maintain. To force explicit declaration of variables, include the statement Option Explicit at the beginning of each script block.

The following block of code illustrates the declaration of several variables in VBScript:

```
...

Dim NewName

Dim ReturnValue, CalculatedValue
```

```
...
```

The scope of variables within an HTML document is discussed later in this chapter, in the section "VBScript Scoping Rules in HTML."

Tip

During this discussion of variables, no mention has been made so far of *constants*. This is for a good reason; there is no special provision for constants in VBScript. To implement constants, simply use standard variables. It is recommended, however, that you adhere to a specific naming convention when using a variable as a constant, such as ensuring that the entire variable name is capitalized, so that constants can be easily distinguished from regular variables.

Variable Assignments

To assign a value to a variable, a single equal sign is used, in the format <*variable*> = <*value*>.

Interestingly, unlike most other languages, in VBScript the assignment operator is the same as the equivalency operator. The following block of script illustrates several different data types being assigned to variables:

```
...

Dim SomeValue, ReturnDate

SomeValue = 32.22     'Begin with a floating point.

SomeValue = "Howdy"   'Since this is a variant, we can

                      'dynamically change its type.

ReturnDate = "9/1/98"

...
```

Arrays in VBScript

This discussion of variables in VBScript ends with the creation and assignment of arrays in VBScript. Arrays are declared using either the Dim or ReDim statement, as shown in the following block of code:

```
...

Dim Lookup(30), AnotherLookup(10,10,2)  'Static arrays
```

```
Dim Reference()       'Dynamic array declaration

Redim NewReference()  'Another Dynamic array declaration

...

Redim Reference(30), NewReference(10)  'Set a new size to these dynamic arrays

...

Redim NewReference(10)        'Set a new size for this array

Redim Preserve Reference(40)  'Set a new size for this array,

                              'preserving its previous contents

...
```

Referring to the preceding snippet, note the following points:

- The first Dim statement declares two arrays, Lookup and AnotherLookup. The size of these arrays cannot be changed.
- The array AnotherLookup is given three dimensions. Arrays may be given a maximum of 60 dimensions.
- The second Dim statement declares a dynamic array called Reference. Subsequent ReDim statements may be used to resize the array.
- The last statement resizes the array Reference to 40 elements. The Preserve keyword forces the original contents of the array, elements 0 to 29, to be retained.

Note

In all arrays, each dimension begins with element zero (0). In the example Dim Test(5), *six* indices are actually created, from Test(0) to Test(5). As a further note, the Microsoft online documentation for VBScript, as of this writing, *incorrectly* states that the upper bound of a dimensioned array is equal to the specified array size *minus* one.

Tip

The variant subtype of an array is the same for all elements in an array. To determine the subtype of an array, you can use the VarType() function, and subtract 8192. This seemingly arbitrary subtraction will yield the "base" subtype number, as described earlier in Table 8.1.

Operators

VBScript provides all of the standard operators that you would expect in a development environment. Tables 8.3 through 8.5 list all operators used in VBScript, listed in order of precedence.

Table 8.3. Arithmetic operators.

Operator Name Example ^ Exponentiation A = B ^ 2 - Unary Negation A = –-B
Places the negation of B into A * Multiplication A = B * 33 / Division A = B / 2 \ Integer Division A = B / 2
Stores the Integer portion of
the division in A Mod Modulo A = B Mod 2
Stores the remainder of B/ 2
in A + Addition A = B + C - Subtraction A = B – C & String Concatenation A = B & C
Use of Addition operator
obtains the same result; use &
for code clarity.

Table 8.4. Comparison operators.

Operator Name Example = Equality If A = B then print "Equal" <> Inequality If A <> B then print "Not Equal" < Less Than If A < B then print "Less Than" > Greater Than If A > B then print "Greater Than" <= Less Than Or Equal To If A <= B then print "LTOE" >= Greater Than Or Equal To If A >= B then print "GTOE" Is Object Equivalency If A is B then print "Same"
Used to determine if two variables refer to the same object.

Table 8.5. Logical operators.

Operator Name Not Logical negation And Logical conjunction Or Logical disjunction XOr Logical exclusion Eqv Logical equivalence Imp Logical implication

Procedures: Subprograms and Functions

Procedures in VBScript, like those found in other popular programming languages, fall into two categories: subprograms and functions.

Procedure names adhere to the same rules as variable names. That is, they must begin with an alphabetic character, cannot contain embedded periods, and cannot exceed 255 characters in length. Procedures must be defined before they are referenced.

Note

Procedures in VBScript always pass variable parameters by value, not by reference.

Subprograms

Subprograms are declared using the Sub...End Sub statements. Immediately following the subprogram name are zero or more parameters, as shown in the following example:

```
...
Sub PrintStatus
     ...
End Sub
...
Sub NewResult( EnteredVal, Multiplier, ReturnedVal )
     'This subprogram multiplies EnteredVal by Multiplier, and
     'returns ReturnedVal.
     ReturnedVal = EnteredVal * Multiplier
End Sub
...
```

To call a subprogram, simply use the name of the subprogram in your code, such as:

```
...
NewResult A, B, C    'One way to call a subprogram
Call NewResult (A, B, C)  'Using the Call statement
...
```

Note in the preceding block of code that when the Call statement is used, parentheses around the parameter list are required.

Functions

Functions are declared using the Function..End Function statements. Functions differ from subprograms in that they return a variant value. The following block of code illustrates the declaration of a function:

```
...
Function CalcAmtDue( Trans() )
     Dim i, SubTotal, Tax
```

```
    TAX = 0.4
    For i = 1 to Ubound( Trans )
         SubTotal = Subtotal + Trans
    Next
    CalcAmtDue = SubTotal + (SubTotal * TAX)   'Required:  Set value of our
                                               'function name before leaving.
End Function
...
```

Note that the most critical statement in the preceding block of code is the assignment of the function name, CalcAmtDue, to a value before termination of the function.

To use a function in VBScript, simply use the function name as an expression in any statement, such as:

```
NewAmount = CalcAmtDue( Trans() )
```

or:

```
If CalcAmtDue( Trans() ) > 500 Then Print "Present Credit Application"
```

Tip

To exit a function prematurely, before the End Function is encountered, use the Exit Function statement. Similarly, to terminate a subprogram before the End Sub, use the Exit Sub statement.

Conditional Execution

Like other languages, VBScript includes all of the conditional execution statements that are required in a structured development environment.

If...Then...Else

The first conditional execution statement that this section briefly discusses is If...Then...Else. The general syntax for

this statement follows:

```
If expr Then
     ... Execute if expr is True
[Else
     ... Execute if expr is False (optional)]
End if
```

The following block of code illustrates some real-world If...Then...Else statements:

```
...
If AmtDue > CreditLine Then
     If (CreditHistory = GOOD_CREDIT) AND (CurrentEmployment = GOOD_EMP) Then
          CreditLine = CreditLine + AmtDue
     Else
          CompleteTrans = False
     End if
Else
     CompleteTrans = True
End if
...
```

Select Case

A second conditional execution statement, useful in many situations when If...Then...Else blocks would prove to be cumbersome and difficult to read, is the Select Case statement, similar to the switch statement in C. A brief grammar for this statement follows:

```
Select Case expr
     Case expr1
```

```
          ... Execute this code if expr = expr1
     [Case expr2
          ... Execute this code if expr = expr2]
     [Case exprn
          ... Execute this code if expr = exprn]
     [Case Else
          ... Execute this code if above Cases do not
          ... evaluate to True]
End Select
```

A real-world example of a Select Case statement in action follows:

```
...
Select Case ActionFlag
     Case 0
          MsgBox "Delete."
     Case 1
          MsgBox "Add."
     Case 2
          MsgBox "Edit."
     Case Else
          'If it is not <= TriggerPrice, then it is > TriggerPrice
          MsgBox "Invalid action specified."
End Select
...
```

Program Flow Control

VBScript provides three essential looping mechanisms, two of which will be discussed in this section:

- For...Next, allowing code to be repeated a fixed number of times.
- Do...Loop, providing both *post-test* and *pretest* conditional looping capabilities.

A third conditional looping statement available in VBScript, While...Wend, will not be discussed in this section, because Do...Loop provides all of the capabilities of the While...Wend statement, and more.

For...Next

To repeat a body of statements a fixed number of times, use the For...Next statement. The general syntax for this statement is:

```
For var = beginval To endval [Step stepval]
    ... Insert statements to be repeated here
Next
```

The following code snippet illustrates nested For...Next statements initializing the values of a two-dimensional array:

```
...
For i = 0 to 9
     For j = 0 to 19
          Array(i, j) = i * j
     Next
Next
...
```

Tip

To exit out of a For...Next statement prematurely, execute the Exit For statement.

Do...Loop

The Do...Loop statement allows a block of VBScript to be executed until a stated condition evaluates to True. The conditional evaluation may occur before the VBScript block is executed in a *pretest* loop, or after it is executed at least once in the form of a *post-test* loop.

The *pretest* form of Do...Loop follows. Note that in this form it is possible that the script sandwiched between the Do and Loop statements may never be executed, depending on the evaluation of the initial expression:

```
...

Do [{While|Until} expr]

    ... VBScript block to execute while expr is True

Loop

...
```

An example of this form of a Do...Loop is shown in this VBScript example:

```
...

NewVal = 0

Do Until NewVal > 100

      NewVal = NewVal + CSng( InputBox ("Enter a new value to add.") )

Loop

MsgBox "The value is: " + CStr( NewVal )

...
```

The *post-test* form of the Do...Loop statement, which allows VBScript within the structure to be executed at least once, is shown following:

```
...

Do

    ... VBScript block to execute while expr is True

Loop [{While|Until} expr]

...
```

Here is an example in VBScript of a *post-test* Do...Loop statement:

```
...

NewVal = 0
```

```
Do

      NewVal = NewVal + CSng( InputBox ("Enter a new value to add.") )

Loop Until NewVal > 100

MsgBox "The value is: " + CStr( NewVal )

...
```

Note

It is possible in VBScript to create an endless loop using the Do...Loop statement. Before distributing your VBScript-enabled Web pages to the world, test your code carefully to ensure that you won't lock up the browsers of surfers in other countries, potentially causing an international incident.

Basic Input/Output

VBScript provides two extremely useful mechanisms for both displaying dialog boxes and requesting input from the user. These built-in functions are described in this chapter.

Displaying Simple Messages with MsgBox

The MsgBox() function allows a standard dialog box to be displayed. If required, the button clicked to close the dialog box may be determined. The syntax for MsgBox is

```
MsgBox(prompt[,buttons][,title][,helpfile,context_id])
```

In this syntax, the parameters are

- *prompt* is the text that you would like to be displayed in the body of the dialog box.
- *buttons* indicates the buttons, dialog box icon, and modality of the dialog box. See Table 8.6 for a list of values to use with this parameter.
- *title* is the text that is displayed in the caption bar.
- *helpfile* is the name of the local help file that should be opened if the user presses F1.
- *context_id* is the help context ID associated with the dialog box with respect to the specified help file.

Table 8.6 lists the options that are summed to produce the *buttons* parameter, defining the general characteristics of the MsgBox() function.

Table 8.6. Add values of desired attributes for *buttons* parameter.

Value Result Button Selection—Default: Ok 0 Ok button is displayed. 1 Ok and Cancel buttons are displayed. 2 Abort, Retry, and Cancel buttons are displayed. 3 Yes, No, and Cancel buttons are displayed. 4 Yes and No buttons are displayed. 5 Retry and Cancel buttons are displayed. *Default Button Selection—Default: First* 0 Assume the first button to be the default button. 256 Assume the second button to be the default button. 512 Assume the third button to be the default button. 768 Assume the fourth button to be the default button. *Dialog Box Icon—Default: None* 16 Use the Critical Message icon. 32 Use the Warning Query icon. 48 Use the Warning Message icon. 64 Use the Information Message icon. *Modality—Default: Application Modal* 0 Create an Application Modal dialog box. 4096 Create a System Modal dialog box; suspend all applications until the dialog box is closed.

If you are required to determine the button pressed by the user, the returned value from MsgBox() is provided. To determine the button selected, use Table 8.7.

Table 8.7. Return result of MsgBox().

Value: Selected Button: 1 Ok 2 Cancel (or ESCape) 3 Abort 4 Retry 5 Ignore 6 Yes 7 No

The following section of VBScript illustrates some examples of MsgBox() in use:

```
...
'In this example, we'll present a dialog box without
'concerning ourselves with overriding the default
'characteristics of the MsgBox.
MsgBox("This is a simple dialog box.")
'Now we'll display a message box that contains
'both Yes and No buttons (4), and a Query icon (32).
'We will also utilize the returned value.
ret = MsgBox("Do you wish to continue?", 4+32)
If ret = 7 Then
     'No
     Exit Sub
End If
...
```

Figure 8.1 shows an example of a simple *MsgBox()*.

Figure 8.1. Simple MsgBox() example that uses Yes/No Buttons and a Query icon.

Requesting Simple Input with InputBox

The InputBox() function, like MsgBox(), is also a built-in routine that makes simple user interaction possible. Whereas MsgBox() is used for simple button selection and user notification, InputBox() allows a text entry to be made by the user. The syntax for InputBox() is shown following:

```
InputBox(prompt[,title][,default][,xpos][,ypos][,helpfile, context_id])
```

In this syntax, the parameters are:

- *prompt* is the text that you would like displayed in the body of the dialog box.
- *title* is the text displayed in the caption bar.
- *default* is a default text value.
- *xpos*, *ypos* are the horizontal and vertical starting positions of the dialog box, respectively. These values are always given in twips. By default, the dialog box will appear centered horizontally, and just above center vertically.
- *helpfile* is the name of the local help file that should be opened if the user presses F1.
- *context_id* is the help context ID associated with the dialog box with respect to the specified help file.

The return value of InputBox() represents the value typed in by the user, or an empty string ("") if Cancel was selected.

Here is an example of the InputBox() function being used in VBScript:

```
...
response = InputBox("Enter first and last name.", "Name Entry")
If response = "" Then
     'Cancel was pressed
     MsgBox ("Hey, you pressed Cancel!!")
End If
...
```

This example code produces the dialog box shown in Figure 8.2.

Figure 8.2. Simple InputBox() example.

Important Built-In Functions

Several important functions built into VBScript that allow for string manipulation and date/time manipulation have not been discussed in this chapter. Table 8.8 lists the most important string manipulation functions that will more than likely be used in any VBScript code that you develop. Instead of providing the syntax for each function, an actual example of how the function is used is shown.

Table 8.8. String manipulation functions.

Example Returns Explanation InStr("To All", "All") 4 Used to find the character position of the first occurrence of the second string in the first string. Left("To Whom It", 2) "To" Returns the specified number of characters from the given string, starting with the leftmost character. Right("To Whom It", 2) "It" Returns the specified number of characters from the given string, starting with the rightmost character. Len("Jordan") 6 Returns the length of the given string. LTrim(" Some ") "Some " Strips all leading spaces from the given string. RTrim(" Some ") " Some" Strips all trailing spaces from the given string. UCase("Kerrie") "KERRIE" Capitalizes all characters in the given string. Mid("Some Junk", 3, 2) "me" In this example, returns two characters starting with the third character.

Comparing VBScript to Visual Basic

For those developers already familiar with Visual Basic or Visual Basic for Applications (VBA), the use of VBScript will prove to be extremely straightforward. VBScript is simply a subset of Visual Basic; its primary design goal was to provide a "light" interpreted language that provides all of the fundamental features, but none of the icing, of Visual Basic.

Some of the most significant differences between the two languages are described following:

- In VBScript, constants (CONST) cannot be declared, nor are any intrinsic constants available, such as those used in the parameter list of MsgBox() and InputBox().
- All arrays declared in VBScript are always based at index 0. The statement Dim Test(5) actually produces six indices, from Test(0) to Test(5). The Option Base directive is unavailable.
- The statements Goto, Gosub, and Return are no longer provided. Consequently, labels are not allowed. This is probably a desirable omission, as it forces more structured development.
- All parameters passed to procedures are passed by value, not by reference. There is no way to alter this behavior. This implies that variables passed to procedures cannot be changed in the procedure itself, unless they are globally declared.
- The Format function, used to format numeric, date/time, or text values, is not provided. For some, this exclusion might present significant opportunities when presenting data to users in a formatted, consistent fashion.
- [1b] In For...Next statements, the VBScript interpreter will trigger an "Expected end of statement" error if a variable name is specified in the Next statement, such as Next i. The Next statement in VBScript should be used by itself, without a variable specification.
- All functions that provided access to external functions or files, such as Windows API calls or file input/output, have been removed. This was necessary to eliminate the possibility of rogue VBScript applications.

Tip

For more information on the differences between the two languages, see http://www.microsoft.com/vbscript/us/vbslang/vsgrpNonFeatures.htm.

In addition to language differences, the development environment (IDE) of VBScript is a throwback to the days of

using an ASCII editor to create and debug your code. The development process, when using VBScript in an HTML document, usually consists of these steps:

1. 1. Use an ASCII editor to modify an HTML document.
2. 2. Fire up your Web browser and load the HTML document.
3. 3. If you find any problem, in the form of either syntax errors or logical errors return to Step #1.

Tools such as the Microsoft ActiveX Control Pad make this process much simpler, especially when integrating ActiveX controls into your HTML document. However, even with this tool, for those who are accustomed to syntax highlighting, automatic syntax checking, and an integrated debugger with stepping, breakpoints, and watch variable facilities will be dissatisfied with the current VBScript development environment options.

VBScript and HTML: On the Rocks

Thus far this chapter's discussion has been based on VBScript itself, with little mention of the use of VBScript in an HTML document. The remainder of this chapter emphasizes how VBScript is utilized in HTML.

How to Embed VBScript into HTML

VBScript is included in HTML using the <SCRIPT></SCRIPT> tag. The following example illustrates a skeleton block used to embed VBScript in HTML:

```
...

<BODY>

<SCRIPT LANGUAGE = "VBScript">

<!--

     'Insert your VBScript statements here

     'This will include variable declarations, procedures and directives

-->

</SCRIPT>

</BODY>

...
```

Please note the following points:

- The LANGUAGE attribute in the <SCRIPT></SCRIPT> block is assigned to "VBScript" using the LANGUAGE attribute.

- Within the SCRIPT tag, VBScript should be placed in an HTML comment block <!-- ... -->, allowing browsers that do not understand the SCRIPT tag to correctly parse the remainder of the HTML document without displaying your VBScript code.

It is recommended that all script be included in one contiguous <SCRIPT></SCRIPT> block embedded in either the <HEAD></HEAD> section or the end of the <BODY></BODY> section of a given HTML document. Although VBScript will not be affected by its placement, it is best, for readability and maintenance purposes, to group all script together in one consistent location.

VBScript Scoping Rules in HTML

There are two levels of scope defined in VBScript that affect the visibility and lifetime of variables: *script* and *procedure*.

The script level of scope involves all VBScript statements that are *not* included in any procedures (functions or subprograms), but that are included in the current HTML document, in any <SCRIPT></SCRIPT> blocks. Any variables declared at the script level can be seen by any script included at the procedure level. Variables are eliminated from memory when the current script is terminated by loading a new HTML document.

The procedure level of scope includes all variables declared within a procedure. These variables are eliminated from memory when the procedure in which they are declared is terminated by either an Exit or End statement.

Tip

The use of these scopes implies that there is no way to retain the value of variables between several HTML documents. If you require this capability, use HTML Cookies.

When Is VBScript Executed?

It is important to remember that VBScript is interpreted "on the fly", as your browser that supports VBScript, such as Microsoft Internet Explorer 3.0, parses through an incoming HTML document. Therefore, include VBScript that is required to immediately modify the attributes of a document at load-time in your <HEAD></HEAD> section, outside of any procedures. Including this code in the <HEAD></HEAD> section is simply a common convention, not a necessity; it could be placed in the <BODY></BODY> sections, but this is not recommended.

Any declared procedures are interpreted and stored in the client's local address space, ready for later execution. Any code at the script level (not embedded in procedures) is executed as encountered while an HTML document is parsed.

As in most other modern-day development environments, all non-script level procedures are executed based on the event-driven model. That is, a procedure is "triggered" based on an action that the user or some other entity (the object model, for example) triggers.

The next section begins to explain how objects and their actions are tied to VBScript procedures.

VBScript and the Internet Explorer Object Model

The Internet Explorer (IE 3.0) Object Model allows VBScript to be used to programmatically control both the current HTML page and the Web browser itself. For example, VBScript and the IE Object Model can be used to:

- Present a dialog box to the user using the alert method defined by the window object.
- Specify and load a page pointed to by a specified URL using the open method defined by the window object.
- Change the background color of the current document using the bgColor property of the document object.

This object model, along with the programmatic power of VBScript, brings a tremendous amount of flexibility to the client side of any HTML document.

Tip

A significant portion of development in VBScript and HTML will require interaction with the IE 3.0 object model. For more information on this technology, refer to Chapter 7, "Microsoft Internet Explorer 3.0 and Its Scripting Object Model."

Using VBScript with an HTML Button

Through the IE 3.0 Object Model, some HTML elements, such as buttons and forms, may trigger the execution of specified VBScript procedures. In this section, you begin to create a real-world HTML page that demonstrates the initiation of a VBScript procedure based on an event.

The first example will enable the user to toggle the caption on an HTML button to read either "On" or "Off". Please refer to Listing 8.1; a complete explanation follows.

Listing 8.1. A simple HTML page that utilizes VBScript.

```
<HTML>

<HEAD>

<H2>

This page uses an HTML-defined button in a form to trigger

a VBScript subprogram, Button_Click.

</H2>

<Form Name = "frmTest">

        <!--

        Define a button called "button" whose initial caption
```

```
        will read "Try Me", and will call the VBScript
        subroutine Button_Click when the onClick event is
        triggered.  Remember that we're still in HTML definition
        here!
        -->
        <Input Type="Button" Name="Button" Value="Try Me"
                onClick="Button_Click" Language="VBScript">
</Form>
<SCRIPT LANGUAGE="VBScript">
<!--
Dim Toggle      'This global variable will retain our current
                'toggle state
Sub Button_Click
        'Now change the caption of the button defined on
        'frmTest to indicate the current value.
        If Toggle = 0 then
                Document.frmTest.Button.Value = "On"
                Toggle = 1
        else
                Document.frmTest.Button.Value = "Off"
                Toggle = 0
        end if
End Sub
-->
</SCRIPT>
<TITLE>Test VBScript!</TITLE>
```

```
</HEAD>

</HTML>
```

This HTML page performs the following tasks:

- A form on an HTML page is created, frmTest, which consists of a single button, referred to as Button.
- The onClick attribute of the button points to the VBScript procedure that will handle the button's click event, Button_Click. The language attribute specifies VBScript as the language used in the event handler.
- The SCRIPT block in the HTML document contains the Button_Click event handler.
- When the user clicks on the button, you simply modify the Value attribute to indicate the current toggle state, either "On" or "Off". Note the use of the Document object, from the IE 3.0 Object Model.

This example illustrates, at a rudimentary level, how the IE 3.0 Object Model allows VBScript to be used as the event handler for an HTML-defined button.

The purpose of this section was to illustrate the use of VBScript in an HTML document without the use of ActiveX controls. However, as the thrust of this book revolves around ActiveX, this chapter will not elaborate on the use of VBScript to control standard HTML elements; this topic falls outside of the scope of this chapter. The next section continues the discussion of VBScript, where you will utilize ActiveX components in HTML.

Using ActiveX and VBScript in HTML

As stated previously, VBScript can be used to programmatically define how an object on an HTML document must react to a triggered event. In the previous section, you examined how VBScript might be integrated along with a standard HTML object, a button in your example, to control what occurs when the user clicks on a button.

In this section, you will utilize readily available ActiveX components into an HTML document and react to various component-related events using VBScript.

Adding an ActiveX Control: The Hard Way

To add an ActiveX control to an HTML document is only slightly similar to adding a standard HTML element, such as a button, to an HTML document.

ActiveX controls and their associated properties are embedded in an <OBJECT></OBJECT> tag. The specified ActiveX control will appear in the HTML document wherever this tag occurs. The following code snippet illustrates the definition of an ActiveX control, in this case an IE 3.0 Label, one of the controls that ship with Microsoft Internet Explorer 3.0:

```
<OBJECT ID="IeLabel1" WIDTH=265 HEIGHT=70

 CLASSID="CLSID:99B42120-6EC7-11CF-A6C7-00AA00A47DD2">

    <PARAM NAME="_ExtentX" VALUE="5609">
```

```
    <PARAM NAME="_ExtentY" VALUE="1482">
    <PARAM NAME="Caption" VALUE="This is a Caption!">
    <PARAM NAME="Angle" VALUE="10">
    <PARAM NAME="Alignment" VALUE="4">
    <PARAM NAME="Mode" VALUE="1">
    <PARAM NAME="FillStyle" VALUE="0">
    <PARAM NAME="FillStyle" VALUE="0">
    <PARAM NAME="ForeColor" VALUE="#000000">
    <PARAM NAME="BackColor" VALUE="#C0C0C0">
    <PARAM NAME="FontName" VALUE="Arial">
    <PARAM NAME="FontSize" VALUE="12">
    <PARAM NAME="FontItalic" VALUE="1">
    <PARAM NAME="FontBold" VALUE="1">
    <PARAM NAME="FontUnderline" VALUE="0">
    <PARAM NAME="FontStrikeout" VALUE="0">
    <PARAM NAME="TopPoints" VALUE="0">
    <PARAM NAME="BotPoints" VALUE="0">
</OBJECT>
```

From the preceding HTML, please note the following points:

- The opening <OBJECT> tag specifies the ID of the object, which represents the name that the remainder of the HTML page will use to refer to this object. The recommended width and height are also specified.
- The CLASSID, or "CLSID", specifies the identification of the ActiveX control, as designated in your system registry.
- The remainder of the lines, before the closing </OBJECT> tag, specify override values for several parameters, or properties, of the control.

The problems with adding ActiveX controls to HTML pages without the assistance of any other utility should be fairly clear to those with experience with a "point and click" development environment such as Visual Basic. These problems are summarized following:

- How can you easily determine the CLASSID? For each ActiveX control, you need to know the full CLSID that must be specified in the CLASSID attribute of the OBJECT tag.
- How do you know what properties (parameters) apply to the object that you're using? ActiveX controls typically have tens of properties, or parameters, that can be overridden in the OBJECT tag using the PARAM NAME

attribute. How can you easily determine what properties are available when using a text editor to create your HTML page?

- How can you determine what methods, or events, a control triggers? Remember, you're using a straight text editor in most cases to design HTML pages, without integrated help.

One possible solution that applies even with add-on development tools might be to browse through Web-based documentation that describes properties and methods for each object. Microsoft's Web site contains excellent descriptions of each of the ActiveX controls bundled in IE 3.0. However, this still does not approach the ease of development found in environments such as Borland Delphi or Microsoft Visual Basic.

The solution to most, but not all, of these dilemmas is described in the next section.

Making Life Easier with Microsoft ActiveX Tools

A good friend of mine once said that, by nature, developers are lazy people. When a programmer finds a task difficult to accomplish, he (or she) will either seek a tool that someone else has developed to help get the job done more quickly or develop a tool themselves that shaves either milliseconds or days off a programming task.

This is a good thing. Microsoft, the makers of the ActiveX technologies that make programmers' lives so very interesting, have also released two invaluable tools that dramatically assist in the process of using ActiveX controls in HTML documents: The ActiveX Control Pad and the ActiveX Control Lister. Both of these tools can be found in the ActiveX SDK.

Tip

If you do not have access to this SDK, visit the Microsoft ActiveX SDK Web site at http://microsoft.com/intdev/sdk. Follow the links to the download area for more details.

ActiveX Control Lister

As mentioned earlier, any time an ActiveX control is added to a Web page using the <OBJECT> tag, the class ID is required. Normally the determination of this value would require the perusal of the Windows system registry.

The ActiveX Control Lister makes this process much simpler by presenting a dialog box, after scanning your system registry, that contains all of the ActiveX controls installed on your system. Additionally, you can right-click on any of the controls in this list box to bring up a pop-up window that will allow you to copy the essentials for the <OBJECT> tag, including the CLSID, into the Windows clipboard for easy pasting into a text HTML file. What a lifesaver!

Figure 8.3 represents a typical ActiveX Control Lister display, along with the pop-up window that appears when the right mouse button is clicked.

Figure 8.3. The ActiveX Control Lister in action.

When Copy is selected from the pop-up menu, the minimal <OBJECT> tag requirements for the selected control are pasted into the clipboard. For example, the following text was pasted into the clipboard when the Microsoft Forms 2.0 Image ActiveX control was selected:

```
<Object

     Id="Microsoft Forms 2.0 Image"

     Classid="clsid:4C599241-6926-101B-9992-00000B65C6F9"

     Width=80

     Height=30 >

</Object>
```

If you select Options from this pop-up menu, you can configure the contents and format of the information that is copied into the clipboard, such as character case, values for the initial width and height attributes, and whether or not the text should appear on one line or multiple lines.

This utility is a "must have" for any Web page developer who utilizes ActiveX controls.

ActiveX Control Pad

The ActiveX Control Pad is described in detail in Chapter 11, "Using ActiveX Control Pad," so this section doesn't go into any significant detail regarding this tool. However, you should understand the fundamental operation of this utility, especially as it applies to VBScript.

At first glance, it may appear as though the ActiveX Control Pad is simply a text editor, with perhaps the ability to automatically present a basic HTML framework in new HTML documents. However, it is much, much more than that. In essence, the ActiveX Control Pad provides the developer with an HTML "Integrated Development Environment" that is especially useful when integrating ActiveX controls, the Internet Explorer 3.0 Object Model, and VBScript or JavaScript.

Here is a brief list of the most important features found in this product:

- Multiple HTML documents may be loaded at the same time in its MDI framework.
- Properties for ActiveX controls may be easily modified through the ActiveX Control Editor, a part of the ActiveX Control Pad. This editor is similar to control editors found in environments such as Visual Basic and Delphi, allowing properties for the chosen control to be easily viewed or modified.
- Once a control is inserted, the Control Pad automatically handles the HTML dirty work, including complete contents of the <OBJECT></OBJECT> block. If you need to modify the properties of an object, simply click on a button that appears to the left of each opening <OBJECT> tag to redisplay the ActiveX Control Editor.
- The Script Wizard, perhaps the most important product feature, allows you to view your HTML document in an "object-oriented" fashion. All ActiveX objects and associated events in the current page appear in a neatly organized dialog box. Either VBScript or JavaScript may be inserted to handle tasks as necessary.

Note

Another important attribute of this product, HTML Layouts, will not be discussed in this section, but is discussed in Chapter 11. HTML Layouts give the Web page designer much more control over how controls will be positioned on the screen.

Figure 8.4 illustrates a typical operation utilized in the ActiveX Control Pad: changing properties for an object, in this case a label, using the Control Editor.

Figure 8.4. The ActiveX Control Pad's Control Editor.

In some cases, a picture is definitely worth a thousand words. Compared to a straight text editor, the advantage of using the ActiveX Control Pad to modify your HTML documents should be apparent.

Of course, simply inserting ActiveX controls into your Web documents and manipulating their properties is only one quarter of the battle. The remaining three quarters revolve around controlling these controls, and allowing them to interact with each other through VBScript or JavaScript.

The ActiveX Control Pad, through the Script Wizard, gives you an easy way to:

- Modify the parameters of controls (objects) already added to your document, supplementing the use of the Control Editor.
- View all of the available objects and associated events associated with each ActiveX object that has been added to the current HTML page.
- View all procedures (VBScript or JavaScript) that have been defined in all HTML script blocks, as well as available methods for all object blocks, including objects defined in the IE 3.0 Object Model.
- Add event handlers for any one of the predefined events that are listed under each object that has been added to the current HTML document.
- Link events to either available methods that apply to the IE 3.0 Object Model, or custom made subprograms.

Figure 8.5 illustrates an event handler being designed for the Click event of the lblWelcome object, an ActiveX label control. Note the other objects and events that are displayed in the upper left panel, and the possible actions listed in the upper right panel.

Figure 8.5. The Script Wizard in the ActiveX Control Pad.

Now that you have a rudimentary understanding of what you can accomplish with the ActiveX Control Pad, this chapter on VBScript will wrap up by designing an interactive Web page that utilizes ActiveX Controls with underlying logic created in VBScript.

An Interactive Page Using VBScript and ActiveX Controls

In this section, you create a Web page that utilizes several ActiveX controls, tying together those controls and controlling some elements of the IE 3.0 Object Model through VBScript. The interactive page that will be demonstrated here is a typical user entry screen that might appear when a Web surfer is requesting information on one or more products from a typical Web site.

For those who have developed interactive forms in standard HTML, you may ask why the example uses ActiveX labels, edit boxes and other controls over elements that are built into the HTML standard. This is for two reasons. First, the purpose of this book is to discuss ActiveX controls, not necessarily built-in HTML elements. Second, ActiveX controls give the developer much more flexibility with respect to their attributes than built-in HTML elements.

Note

The CD-ROM included with this book contains the HTML source for each phase of development included in this section, beginning with INTER.HTM, then following with INTER2.HTM, and INTER3.HTM.

Note

The goal of this section is to illustrate the use of VBScript fundamentals described in the first half of this chapter. Therefore, I will not discuss the use of the ActiveX Control Pad, which was used to generate these pages.

The Framework

The first phase in designing any HTML page is to lay out the basic components of the page. You'll use VBScript and the methods available in the IE 3.0 Object Model to create your basic framework, shown in Listing 8.2.

Listing 8.2. The "Basics"—Your Framework

```
<HTML>

<HEAD>

<TITLE>Sample Interactive Page</TITLE>

<SCRIPT LANGUAGE="VBSCRIPT">

<!--

'This script is executed immediately, when the form is loaded...

Dim TodayNum

'Change the background color to white

window.document.bgColor = "#FFFFFF"

'Print the name of the current day

TodayNum = Weekday( Date )

window.document.write("<H3>It is  "&NameOfDay(TodayNum)&" at "&Time&"</H3>")

'Create a new paragraph, and add it to the current document:

window.document.write("<P><H4>")

window.document.write("Thank you for requesting information on our products!")

window.document.write("Please carefully fill out each entry on this form.  ")

window.document.write("When you are finished, click on the Submit button.")
```

```
window.document.write("</H4></P>")
Function NameOfDay( DayNum )
     'Given a day number, return the full name of the
     'day; use with WeekDay built-in function.
     Select Case DayNum
     Case 1
          NameOfDay = "Sunday"
     Case 2
          NameOfDay = "Monday"
     Case 3
          NameOfDay = "Tuesday"
     Case 4
          NameOfDay = "Wednesday"
     Case 5
          NameOfDay = "Thursday"
     Case 6
          NameOfDay = "Friday"
     Case 7
          NameOfDay = "Saturday"
     End Select
End Function
-->
</SCRIPT>
</HEAD>
<BODY>
</BODY>
```

```
</HTML>
```

Note the following points about this basic framework:

- The explicitly declared variable TodayNum is a globally accessible variable; it exists at the script level because it was not declared in a procedure.
- The <SCRIPT></SCRIPT> block is contained within the <HEAD></HEAD> HTML block. This is standard HTML scripting practice.
- Instead of including introductory text directly in the HTML code, the example instead used the write method of the Document object. This method is defined by the IE 3.0 object model.
- Instead of using the HTML <BODY> tag to specify a background color, the example wrote directly to the current document using the bgColor property of the Document object. This object is defined in the IE 3.0 object model.
- The function NameOfDay() uses the Select...Case statement to return the string representation of the passed day number. Because this function returns a string, it is used directly in the first window.document.write statement as a string expression.

The image shown in Figure 8.6 depicts your current Web page.

Figure 8.6. Your "base" Web page, without ActiveX Controls.

Adding ActiveX Controls

Now that you have a very primitive framework for your page, you can begin adding several ActiveX controls that will enable the user to enter all the information the program will need to process his request for information. You will also construct a simple HTML table, each cell of which will be used to contain an ActiveX control. Remember that without the use of a table, ActiveX objects will appear in order as they are encountered in your HTML document. Using tables is a simple way to ensure that the controls in your form are properly positioned.

Note

Most of the ActiveX controls used in this example are shipped with the ActiveX Control Pad, and they are not part of the standard Internet Explorer 3.0 control set. Information pertaining to the properties, methods, and events that apply to these controls can be found in the ActiveX Control Pad help system.

Tip

For many ActiveX controls, including the ones shipped with the ActiveX Control Pad, the CodeBase property should be defined, indicating a URL (a Web site, ftp site, or gopher site, for example) where the control can be downloaded. If a Web page is loaded that uses an ActiveX control that the user does not already have on her system, this mechanism will provide a means to retrieve and register the needed control automatically.

Listing 8.3, taken from the <BODY></BODY> portion of your HTML document, contains your basic table and all of the ActiveX controls that will be contained in the cells of the table.

Listing 8.3. Your <BODY> with a table and ActiveX Controls.

```
<BODY>
<TABLE BORDER=0>
<TR>
        <TD>
             First Name:
        </TD>
        <TD>
        <OBJECT ID="txtFirstName" WIDTH=291 HEIGHT=24
         CLASSID="CLSID:8BD21D10-EC42-11CE-9E0D-00AA006002F3">
        <PARAM NAME="VariousPropertyBits" VALUE="746604571">
        <PARAM NAME="Size" VALUE="7694;635">
        <PARAM NAME="FontCharSet" VALUE="0">
        <PARAM NAME="FontPitchAndFamily" VALUE="2">
        <PARAM NAME="FontWeight" VALUE="0">
    </OBJECT>
</TD>
</TR>
<TR>
        <TD>
            Last Name:
        </TD>
        <TD>
        <OBJECT ID="txtLastName" WIDTH=291 HEIGHT=24
        CLASSID="CLSID:8BD21D10-EC42-11CE-9E0D-00AA006002F3">
        <PARAM NAME="VariousPropertyBits" VALUE="746604571">
```

```
        <PARAM NAME="Size" VALUE="7694;635">
        <PARAM NAME="FontCharSet" VALUE="0">
        <PARAM NAME="FontPitchAndFamily" VALUE="2">
        <PARAM NAME="FontWeight" VALUE="0">
    </OBJECT>
</TD>
</TR>
<TR>
        <TD>
            Company:
        </TD>
        <TD>
        <OBJECT ID="txtCompany" WIDTH=291 HEIGHT=24
         CLASSID="CLSID:8BD21D10-EC42-11CE-9E0D-00AA006002F3">
        <PARAM NAME="VariousPropertyBits" VALUE="746604571">
        <PARAM NAME="Size" VALUE="7694;635">
        <PARAM NAME="FontCharSet" VALUE="0">
        <PARAM NAME="FontPitchAndFamily" VALUE="2">
        <PARAM NAME="FontWeight" VALUE="0">
    </OBJECT>
</TD>
</TR>
<TR>
        <TD>
             Mail Address:
        </TD>
```

```
        <TD>
        <OBJECT ID="txtEMail" WIDTH=291 HEIGHT=24
        CLASSID="CLSID:8BD21D10-EC42-11CE-9E0D-00AA006002F3">
        <PARAM NAME="VariousPropertyBits" VALUE="746604571">
        <PARAM NAME="Size" VALUE="7694;635">
        <PARAM NAME="FontCharSet" VALUE="0">
        <PARAM NAME="FontPitchAndFamily" VALUE="2">
        <PARAM NAME="FontWeight" VALUE="0">
    </OBJECT>
</TD>
</TR>
<TR>
        <TD>
            Product Interest:
        </TD>
        <TD>
        <OBJECT ID="lstProducts" WIDTH=203 HEIGHT=76
         CLASSID="CLSID:8BD21D20-EC42-11CE-9E0D-00AA006002F3">
        <PARAM NAME="ScrollBars" VALUE="3">
        <PARAM NAME="DisplayStyle" VALUE="2">
        <PARAM NAME="Size" VALUE="5362;2011">
        <PARAM NAME="ListStyle" VALUE="1">
        <PARAM NAME="MultiSelect" VALUE="1">
        <PARAM NAME="SpecialEffect" VALUE="0">
        <PARAM NAME="FontCharSet" VALUE="0">
        <PARAM NAME="FontPitchAndFamily" VALUE="2">
```

```
        <PARAM NAME="FontWeight" VALUE="0">
    </OBJECT>
</TD>
</TR>
        <TD>
             Level of Interest:
        </TD>
        <TD>
        <OBJECT ID="cmbInterest" WIDTH=203 HEIGHT=24
        CLASSID="CLSID:8BD21D30-EC42-11CE-9E0D-00AA006002F3">
        <PARAM NAME="DisplayStyle" VALUE="7">
        <PARAM NAME="Size" VALUE="5366;635">
        <PARAM NAME="ListRows" VALUE="3">
        <PARAM NAME="MatchEntry" VALUE="1">
        <PARAM NAME="ShowDropButtonWhen" VALUE="2">
        <PARAM NAME="FontCharSet" VALUE="0">
        <PARAM NAME="FontPitchAndFamily" VALUE="2">
        <PARAM NAME="FontWeight" VALUE="0">
    </OBJECT>
</TD>
</TR>
</TABLE>
<PLAINTEXT>
</PLAINTEXT>
<CENTER>
<OBJECT ID="cmdSubmit" WIDTH=159 HEIGHT=49
```

```
 CLASSID="CLSID:D7053240-CE69-11CD-A777-00DD01143C57">
    <PARAM NAME="Caption" VALUE="Submit Information">
    <PARAM NAME="Size" VALUE="4202;1291">
    <PARAM NAME="FontEffects" VALUE="1073741825">
    <PARAM NAME="FontHeight" VALUE="200">
    <PARAM NAME="FontCharSet" VALUE="0">
    <PARAM NAME="FontPitchAndFamily" VALUE="2">
    <PARAM NAME="ParagraphAlign" VALUE="3">
    <PARAM NAME="FontWeight" VALUE="700">
</OBJECT>
</CENTER>
</BODY>
```

Tip

If you're wondering where the author got the PARAM NAME values, remember that this page was designed using the ActiveX Control Pad, which gives the developer all available parameters for each ActiveX control.

With respect to the preceding listing, note the following points:

- The OBJECT ID for each of the ActiveX controls used in your <BODY> is given a name that follows conventions established in the Visual Basic world, whereby the object identifier may be used to indicate the control from which it is based. For example, txtCompany is used as an ID for a text entry box, and lstProducts is used as an ID for a list box.
- The default values for each control in the parameters added through the ActiveX Control Pad were used, with two exceptions. The ListBox control, used to allow one or more products to be selected, had its MultiSelect property changed to 1-Multi, and its ListStyle property set to 1-Option. In addition, the Style property for the combo box control, used to set a level of purchasing interest, was set to 2-DropDownList.
- The controls included in this section of HTML appear as they are encountered by the Web browser, just like all other HTML elements.
- As of now, there is no code behind the controls, so the form essentially does nothing besides display itself when loaded. This will be remedied in the next section.

In Figure 8.7, you can see what your current Web page looks like when viewed through the eyes of a Web browser.

Figure 8.7. Your Web page with several ActiveX Controls.

Controlling Your Controls with VBScript

Now that you've created your primary controls, you need to place VBScript behind them to:

- Add a list of possible products and levels of interest to the list box and combo box controls that you've added to your page.
- Act upon the Tab key so that focus moves to the next control on the page.
- Act upon the Click event for the Submit button. In this case, you need to first validate all user entries, then display a message box to the user allowing her to accept or cancel her submission.

Your first task is to populate both the product list and level of interest list when the Web page is loaded. To do this, you will use the OnLoad event of the Window object, which is part of the IE 3.0 object model. The AddItem method for the two controls allows you to add items to the list box and combo box. The following subprogram should be placed at the end of your existing <SCRIPT> block:

```
Sub Window_onLoad()

     'Load up our list box with default values

     lstProducts.AddItem "Wacky Widgets"

     lstProducts.AddItem "The Roto Box"

     lstProducts.AddItem "Gucci Babies"

     'Load up our combo box with default values

     cmbInterest.AddItem "Purchase w/in a Week"

     cmbInterest.AddItem "Purchase w/in a Month"

     cmbInterest.AddItem "Just Browsing"

end sub
```

The next task, acting upon a Tab key, is fairly simple and does not require the use of VBScript. All that is required is a modification of one of the default parameters for each of the controls that you've added. The following parameter should be added to each text box control; the other controls already include this behavior by default:

```
<PARAM NAME="TabKeyBehavior" VALUE="-1">
```

Your final task, input validation, does, of course, require the inclusion of additional VBScript. You will simply check the value of each of your controls in the Click event of your Submit button (cmdSubmit), ensuring that information has been entered into each field.

The subroutine shown in Listing 8.4 should be added to the end of the existing <SCRIPT> block.

Listing 8.4. Your input validation routines.

```
Sub cmdSubmit_Click()

    Dim i, OneSel, FinalMsg, resp

    'All input validation is done here.

    if Trim(txtFirstName.Value) = "" then

         MsgBox "You must enter your First Name!",48, "Input Error" 'Warning

         exit sub

    end if

    if Trim(txtLastName.Value) = "" then

         MsgBox "You must enter your Last Name!", 48, "Input Error"   'Warning

         exit sub

    end if

    if Trim(txtCompany.Value) = "" then

         MsgBox "You must enter your Company Name!", 48, "Input Error"'Warning

         exit sub

    end if

    if Trim(txtEMail.Value) = "" then

         MsgBox "You must enter your Email Address!",48, "Input Error"'Warning

         exit sub

    end if

    OneSel = False

    For i = 0 to (lstProducts.ListCount - 1)

         If lstProducts.Selected = True then

              OneSel = True

              Exit For

         End If

    Next
```

```
    If OneSel = False Then
         MsgBox "You must select one or more Products!",48 ,"Input Error"
         Exit Sub
    End If
    If Trim(cmbInterest) = "" then
         MsgBox "You must select your Level of Interest!", 48, "Input Error"
         exit sub
    End If
    'If we've made it here, all entries are O.K., so summarize
    'and ask the user if they want to continue...
    'Note Chr(13) moves to the next line in a dialog box...
    FinalMsg = "Please confirm your entries..."+Chr(13)+Chr(13)
    FinalMsg = FinalMsg + "Name: " + txtFirstName + " " + txtLastName + Chr(13)
    FinalMsg = FinalMsg + "Company: "+ txtCompany + Chr(13)
    FinalMsg = FinalMsg + "EMail Address: "+ txtEmail + Chr(13) +  Chr(13)
    FinalMsg = FinalMsg + "Product Interest: " + Chr(13)+Chr(13)
    For i = 0 to (lstProducts.ListCount - 1)
         If lstProducts.Selected = True then
              FinalMsg = FinalMsg + "      " + lstProducts.List + Chr(13)
         End If
    Next
    FinalMsg = FinalMsg + Chr(13) + "Are these correct?"
    resp = MsgBox (FinalMsg, 4+32, "Confirm Entries...")
    If resp = 6 Then 'Yes
         'Submit material - do whatever processing is required in your
         'application to send this information to the server....
```

```
          '...

          'Assuming everything was O.K., send user confirmation

          TodayNum = Weekday( Date )

          document.clear

          document.write

          document.write("<P><H4>")

          document.write("Your submission has been received." )

          document.write("<I>Thank you for your interest in our products!</I>" )

          document.write("" )

          document.write("</H4></P>")

          TodayNum = TodayNum + 1

          if TodayNum = 8 then TodayNum = 1

          document.write("<H3>Your response should be received by: 5:00 PM

               on " & NameOfDay(TodayNum)& "!</H3>")

          document.close

     Else

             MsgBox "You may now modify your entries as necessary.  When you are
[icc]   ready to re-submit, press the Submit button.", 0, "Correct Your Entries"

     End if

End sub
```

With respect to the preceding listing, note the following points:

- For all TextBox controls, the Value property is used to obtain the contents of the box. However, addressing the name of the text box control would result in the same functionality, as value is the default property for this control.
- To determine if at least one product was selected, the program has simply iterated through all of the list items, examining the Selected property. Validation continues if at least one of these values is True.
- If a field does not validate correctly, the Exit Sub statement is executed to simply terminate the validation process. A message box (MsgBox) using the Warning icon (48) is displayed to the user indicating the source of a validation error.
- When all validation is successful, information is summarized and displayed to the user in a message box that contains both Yes and No buttons. If Yes is selected, the current window is cleared, and the user is told that his

request has been processed. In addition, an approximate time and day that a reply can be expected is calculated and displayed. Once again, this is done using the IE 3.0 object model, scripted with VBScript.

This concludes the development of your basic interactive Web page. To complete this page, you might add appropriate code for your particular system to properly submit information to a server, either through CGI or some other server API mechanism.

Summary

VBScript is a fairly limited but functional scripting language that is appropriate for use as a low-overhead Internet scripting tool. As the world of the Internet changes on a minute-to-minute basis, one of the best sources of information will be on the Internet itself, where companies like Microsoft and Netscape share new development strategies and distribute their scripting product directions and documentation. When developing an HTML page using ActiveX controls, you will more than likely find yourself jumping back and forth between your HTML development environment, such as the ActiveX Control Pad, and your Internet browser of choice, allowing you to retrieve the latest technical documentation available on the Web at design time.

The following list summarizes the most important issues discussed in this chapter:

- In its first incarnation, VBScript will be most prevalent on the Internet, used as an ActiveX and general HTML scripting tool.
- The only data type available in VBScript is the variant. Variants are classified into subtypes, which include strings, Booleans, and integers.
- Variables and arrays may be explicitly declared using the Dim statement. Variables used without explicit declarations are declared implicitly. Arrays declared using the ReDim statement are dynamic and may be resized at runtime using subsequent ReDim statements.
- VBScript allows the declaration of two types of procedures: subprograms (Sub) and functions (Function). The only difference between the two is that functions return a variant value. Remember that variables are always passed by value, not by reference, in VBScript.
- VBScript includes numerous program flow control statements, including If..Then..Else, Select Case, For..Next, and Do..Loop.
- Two built-in functions, MsgBox() and InputBox(), may be used to easily display a dialog box to the user, or to request a simple user input.
- For Visual Basic and/or VBA developers, VBScript will be easy to pick up, as it is a subset of these more powerful languages. VBScript removes significant functionality from the more traditional BASIC family. Any functions that may have used a means to destroy or alter the user's environment have been eliminated. In other words, you can't write a virus in VBScript.
- VBScript is included in HTML using the <SCRIPT></SCRIPT> block, with the LANGUAGE = "VBSCRIPT" attribute. VBScript is executed "on the fly" as it is parsed by a Web browser.
- There are two levels of variable scope in VBScript: script and procedure level.
- The Internet Explorer 3.0 Object Model allows direct access to the contents of an HTML page using VBScript. This object model should be studied by anyone who is using VBScript in HTML, as it provides a method to link standard HTML elements, such as buttons, to VBScript procedures using standard event definitions, such as OnClick.
- The ActiveX Control Pad brings welcome relief to the development of HTML creation that utilizes ActiveX Controls. This tool provides scripting abilities and "point and click" control editing that makes ActiveX development for HTML bearable.

- Each ActiveX control has associated with it one or more events that may trigger the execution of VBScript procedures, such as *obj*_OnClick and *obj*_KeyPress. The real power and use of VBScript lies in controlling these embedded ActiveX components, as well as interacting with the IE 3.0 object model.

SAMS Publishing

ActiveX Programming Unleashed, 154-8, 8

filename: axu8MB.DOC

MS Pages: 26
Listing Pages: 12
Figure Count: 7
----------- -----
Total Pages: 41

Last Update: 10/02/96 11:02 AM

Chapter 9

JavaScript

by Rob McGregor

The Web is alive with interactivity these days, and one of the reasons for this is JavaScript. Although ActiveX components are the new kids on the block in bringing interactive content to the Web, JavaScript has already been around the block a few times. JavaScript has the capability to tie together Java applets and ActiveX components with highly interactive Web pages, providing an exciting Web browsing experience.

What Is JavaScript?

JavaScript is a scripting language that enables you to write scripts embedded in HTML pages, and it has features similar to VBScript (which was discussed in Chapter 8, "VBScript"). There are many parallels between the two scripting languages, but the underlying model is quite different. Before looking at just what JavaScript is and how it works, take a look at where it came from.

Origins of JavaScript

What is now known as JavaScript began life as a Netscape scripting language called LiveScript, originally designed for use with the Netscape LiveWire server software. The original idea was that network administrators could use LiveScript on both the server side and the client side. On the server side, administrators would control database connectivity, manage complex web sites, and automate many otherwise tedious administrative tasks. On the client side, Web page authors would be able to provide new and exciting interactive content in HTML code.

The Java Connection

LiveScript was also designed to enable HTML authors to communicate with and control Java applets. Java is a programming language derived from C and C++, yet without many of the shortcomings inherent in these languages. Java provides a fully object-oriented programming model that has built-in security features that are well suited for writing applications and applets for the Internet. The idea was that the combination of Java applets and LiveScript would allow a much richer, much more immersive Web experience for the user.

Enter Sun Microsystems

In cooperation with Sun Microsystems, Netscape hammered the bugs out of LiveScript and folded in more Java-like syntax to their new scripting language. This resulted in the December, 1995, announcement that LiveScript would henceforth be known as JavaScript.

Note

Microsoft Internet Explorer 3.0 supports an implementation of JavaScript, known as JScript, through the Microsoft dynamic link library JSCRIPT.DLL.

Differences Between JavaScript and Java

It's important to realize that Java and JavaScript are two completely separate things. Although there is a similarity in syntax between the two, they are very different. Some important differences between JavaScript and Java are listed following:

- Unlike Java, JavaScript isn't a compiled language; it's interpreted by the client's browser.
- JavaScript uses code that's embedded right into an HTML page, whereas Java applets are compiled separately but can be accessed from within HTML pages.
- JavaScript uses loosely typed, non-declared variable data types, while Java uses strongly typed, declared variable data types.
- JavaScript is object-based, whereas Java is object-oriented. This means that JavaScript code uses built-in, extensible objects, but the notions of classes and inheritance don't exist. Java, on the other hand, uses classes and inheritance in true object-oriented fashion.
- JavaScript checks object references at runtime with dynamic binding, whereas Java objects must exist at compile time and use static binding.

Like all scripting languages, JavaScript has its own syntax, and the next section examines this syntax.

JavaScript Objects

JavaScript borrows much of its syntax from Java, and Java borrows much of its syntax from C. So if you're familiar with Java, C, or C++, JavaScript should look pretty familiar, and it will probably be easier for you to use than VBScript.

A software object typically wraps some concept or functionality into an easy-to-use package. JavaScript, much like Java, expresses everything in terms of objects. An object typically contains *properties*, which define visual characteristics, and *methods*, which define functionality. The JavaScript language is composed of a rather simple hierarchy of two basic types of objects.

- Browser objects, which appear as visible user-interface objects in a browser window.
- Data objects, which are built-in data objects that remain behind the scenes. These objects represent and control certain types of data used by a Web page.

When you load an HTML page into a browser, several objects are created that correspond to the page and its contents. A page *always* has the following objects by default:

- Window
- Location
- History
- Document

In addition to these default objects, your script can specify objects that are created automatically for use by the script when the page loads. The next section explores each of the basic JavaScript objects in greater detail.

JavaScript Events

Events occur in JavaScript in response to user actions in the browser window, and these events are available for various browser objects (described in the next section). The events defined for JavaScript objects are listed in Table 9.1.

Table 9.1. Events defined for JavaScript.

Event Handler Called Just Before onBlur Input focus is removed from a form element. onClick A form element or link is clicked. onChange The value for a text, textarea, or select element has changed. onFocus A form element gets the input focus. onLoad A page is loading in the browser. onMouseOver The mouse pointer moves over a link or anchor. onSelect A form element's input field is selected. onSubmit A form's data is submitted. onUnload A browser page is exited.

Browser Objects

As stated previously, JavaScript browser objects are those that are visible in a browser window. There are five main visible objects that you'll spend most of your time working with in JavaScript, and these are as follows:

- Window object
- Document object
- Form object
- Button object
- Text object

There are many other objects in the hierarchy in addition to the five listed here. These other objects further refine the functionality of the main five, or provide ancillary services that enhance one of the main five objects.

Note

JavaScript uses an *object-based hierarchy*, not a true *object-oriented hierarchy* in the sense of Java or C++. JavaScript uses a simple containment hierarchy, so there can be no inheritance of properties or methods from parent to child as there is in Java and C++.

Take a closer look at each of the browser objects in the hierarchy to see what properties, methods, and events they provide.

Window Object

The Window object is found at the top of the JavaScript hierarchy, and everything occurs within the context of the window. The window contains properties that apply to the entire window, and it holds the document that runs the JavaScript.

Note

There can be more than one window contained within the main window. Each child window in a frames document has a corresponding window object. Frames are discussed in the "Frame Object" section later in this chapter.

Window Object Event Handlers

The window object contains two event handlers that can trigger some action in your scripts: onLoad and onUnload.

- The onLoad handler is triggered when an HTML page is loaded into the browser window.
- The onUnload event is triggered when a user leaves the page.

The onLoad and onUnload event handlers are used within either an HTML <BODY> tag or an HTML <FRAMESET> tag. For example, in a BODY tag these handlers might look like this:

```
<BODY onLoad="..." onUnload="...">
```

The "..." would be replaced by calls to some JavaScript functions (see the section "JavaScript Functions" later in this chapter for more information).

Window Object Properties

The window object contains several properties for use in your scripts, and Table 9.2 lists these properties.

Table 9.2. Window object properties.

Property Description defaultStatus The default string that appears in the window's status bar. frames An array depicting independently scrollable child windows (frames), each with a distinct URL. length The number of frames in a parent window. name The name of a window (represented by the windowName argument). parent A reference to the parent of a frames window. self An implicit parameter passed whenever a method is called, or a property is set, that refers to the same object as a given window object (similar to Me in Visual Basic or this in C++). status A string that appears temporarily in a window's status bar. top A reference to the top (main) window in a frames window relationship. window An expression referring to the windowName argument that refers to the current window.

A window object also contains these other objects as properties:

- document
- frame
- location

Although the defaultStatus property can be set at any time, it's typically done when a page loads. In this case, it's expressed as part of the window.onLoad statement in a <BODY> tag, as follows:

```
<BODY onLoad="window.defaultStatus='Set default text here...'">
```

Window Object Methods

The window object also exposes several methods for use in your scripts, and Table 9.3 lists these methods.

Table 9.3. Window object methods.

Property Description alert() Displays an Alert message box for the user clearTimeout() Cancels a timeout previously set with the setTimeout method close() Closes a window confirm() Displays a Confirm message box with OK and Cancel buttons to let the user make a decision open() Opens a new window prompt() Displays a Prompt dialog box with a message and an edit box that enables the user to input data setTimeout() Waits a specified amount of time before evaluating an expression

Frame Object

The frame object is a property of a window object, and it can be part of a frames array as well. Table 9.4 shows the properties of a frame.

Table 9.4. Frame object properties.

Property Description frames An array depicting all the frames in a window name The NAME attribute of the <FRAME> tag length The number of child frames within a frame parent A reference to the window or frame containing the current frameset self A reference to the current frame window A reference to the current frame

Frames also provide the methods shown in Table 9.5.

Table 9.5. Frame object methods.

Method Description clearTimeout() Cancels a timeout previously set with the setTimeout method setTimeout() Waits a specified amount of time before evaluating an expression

Document Object

A document object is a property of a window or frame object and embodies an HTML document. An HTML document is defined by the current URL and is composed of two sections:

- The *head*, which is defined by the HTML <HEAD> tag. The head contains information defining a document's title and absolute URL base (used for relative URL links within a document).
- The *body*, which is defined by the HTML <BODY> tag. The body defines the entire body of the document, including all other HTML elements for the document.

Note

JavaScript functions are usually placed within the <HEAD> tag to ensure that they are loaded before the body, and that a user won't be able to inhibit JavaScript activity before a script is ready to handle user input.

Document Object Properties

A document object provides many properties for manipulating document data and appearance, and these are listed in Table 9.6.

Table 9.6. Document properties.

Property Description alinkColor Represents the ALINK attribute anchors An array reflecting all the anchors in a document bgColor Represents the BGCOLOR attribute cookie Specifies a cookie fgColor Represents the TEXT attribute forms An array reflecting all the forms in a document lastModified Represents the date a document was last modified linkColor Represents the LINK attribute links An array reflecting all the links in a document location Represents the complete URL of a document referrer Represents the URL of the calling document title Represents the contents of the <TITLE> tag vlinkColor Represents the VLINK attribute

These four JavaScript objects are also properties of the document object:

- anchor
- form
- history
- link

Document Methods

In addition to the properties listed in Table 9.6, the document object also provides the methods shown in Table 9.7.

Table 9.7. Document methods.

Method Description clear() Clears all content from a document window. close() Closes an output stream and displays data sent to the document window using the write() or writeln() methods. open() Opens a stream to collect the output of write() or writeln() methods. write() Writes one or more HTML expressions to a document in the specified window. writeln() Writes one or more HTML expressions to a document in the specified window followed by a newline character.

Location Object

The location object represents a complete URL, and the properties of a location object represent the different elements that make up a complete URL.

Location Object Properties

The location object provides the properties shown in Table 9.8 to provide information relevant to a location.

Table 9.8. Location object properties.

Property Description hash Defines an anchor name in the URL host Defines the *hostname:port* portion of the URL hostname Defines the host and domain name, or IP address, of a network host href Defines the entire URL pathname Defines the *url-path* portion of the URL port Defines the communications port that the server uses for communications protocol Defines the beginning of the URL, including the colon search Specifies a query

These properties are used as elements of a URL as follows:

```
protocol//hostname:port pathname search hash
```

History Object

A history object "remembers" where a user has been during the course of a browsing session. The history object has only one property: length. The length property reveals the number of entries in a history list. History lists are available for windows, frames, and documents, and Table 9.9 lists the methods available for this object.

Table 9.9. History object methods.

Method Description back Performs the same action as when a user clicks the Back button in a browser forward Performs the same action as when a user clicks the Forward button in a browser go Navigates to the location in the history list specified by a numeric parameter or a string parameter

Form Object

A form is a very important construct for JavaScript, and form *elements* enable users to interact with a document through familiar user interface controls. Each form in a document is a distinct object composed of various other objects including

- button
- checkbox
- hidden
- password
- radio
- reset
- select
- submit
- text

- textarea

Table 9.10 shows the properties available for the form object.

Table 9.10. Form object properties.

Property Description action Represents the ACTION attribute elements An array containing all the elements in a form encoding Represents the ENCTYPE attribute length Represents the number of elements on a form method Represents the METHOD attribute target Represents the TARGET attribute

A forms array has a single property, length, which reveals the number of forms in a document's forms array.

A form has a single method, submit, which sends data back to an HTTP server. The submit method returns data using one of two methods: get or post (specified by the method property).

A form also has a single event handler, onSubmit, which occurs when a user submits a form. This results in JavaScript code defined within the event handler being executed.

Text, Textarea, and Password Objects

A text object is a single-line text input box in an HTML form that enables a user to enter text or numbers. Similarly, a textArea object is a multiple-line text input box in an HTML form that enables a user enter text or numbers. A password object is a single-line text input box in an HTML form that enables a user enter text or numbers but masks the values entered by displaying asterisks (*).

Text, Textarea, and Password Properties

All three of these objects are closely related and have the same basic properties and methods. Table 9.11 shows the properties defined for these objects.

Table 9.11. Text object properties.

Property Description defaultValue Represents the text in an object name Represents the name of an object found in the NAME attribute value Represents the current value of the text object's data as set by the VALUE attribute

The defaultValue property is different for each of these objects, as follows:

- For a text object, defaultValue is initially the value of the VALUE attribute
- For a textarea object, defaultValue initially reflects the value specified between the <TEXTAREA> and </TEXTAREA> tags
- For a password object, defaultValue initially is null no matter what value is specified for the VALUE attribute

Text, Textarea, and Password Methods

Table 9.12 shows the methods defined for these objects.

Table 9.12. Text object methods.

Method Description focus() Gives a text object the input focus blur() Removes the focus from a text object select() Selects the input area of the specified text object

Text, Textarea, and Password Event Handlers

There are three event handlers that correspond to the three methods listed in Table 9.12 and one additional method that fires when the value of a text, textArea, or password object changes. Table 9.13 shows the event handlers defined for these objects.

Table 9.13. Text object event handlers.

Event Handler Indicates onBlur A text, textArea, or password object has lost the input focus onChange The data has changed in a text, textArea, or password object onFocus A text, textArea, or password object has received the input focus onSelect Text within a text, textArea, or password object has been selected

Hidden Object

A hidden object is a form element used to pass value/name pairs when a form's data is submitted. As such, a hidden object must be defined within a <FORM> tag.

A hidden object has only two properties:

- name, which represents the name of the object as specified by the NAME attribute.
- value, which represents the current value of the hidden object.

Button Object

A button object is a pushbutton on an HTML form. A button has two properties:

- The name of the button as specified by the NAME attribute.
- The text displayed on the button as specified by the VALUE attribute.

A button has one method, click(), which simulates a mouse click on the button. A button also has a corresponding event handler, onClick, that fires when the button is clicked.

Radio Object

A radio object is an array of mutually exclusive radio buttons on an HTML form that enable a user to choose one item from a list. The radio object has a click() method and an onClick event handler, just like a button object. In addition, a radio object supports the properties listed in Table 9.14.

Table 9.14. Radio object properties.

Property Description checked Specifies that a radio button element is selected defaultChecked Specifies whether a radio button element is initially checked or not, as specified with the CHECKED attribute length The number of radio buttons in a radio object name The name of the radio object as specified by the NAME attribute value The text of the radio object as specified by the VALUE attribute

---

Note

All radio buttons in a group must be referenced by their index numbers within the radio button array.

Checkbox Object

A checkbox object is an HTML form element that can be toggled to checked or not checked (on or off). A checkbox object has a click() method and an onClick event handler, just like a button object. In addition, a checkbox object supports the properties listed in Table 9.15.

Table 9.15. Checkbox object properties.

Property Description checked Specifies that a checkbox is checked defaultChecked Specifies whether the checkbox is initially checked or not as specified with the CHECKED attribute name The name of the checkbox object as specified by the NAME attribute value The text of the checkbox object as specified by the VALUE attribute

Select Object

A select object is an HTML form element that can take two forms:

- A *selection list* on an HTML form, in which case only one element in the list can be selected at a time (analogous to a drop-down list box in a Windows program).
- A scrolling list on an HTML form, in which case a user can choose one or more items from a list simultaneously (analogous to a multiple-selection list box in a Windows program).

Select Object Properties

A select object has the properties listed in Table 9.16.

Table 9.16. Select object properties.

Property Description length The number of options in a select object name The name of the object as specified by the NAME attribute options An array of options a user can choose from as specified by the <OPTION> tag selectedIndex The index of the selected option (or the first selected option if multiple options are selected)

The Options Array

The options array contains an entry for each option in a select object, and the select object's options array itself has several properties. These properties are listed in Table 9.17.

Table 9.17. The properties for an options array.

Property Description defaultSelected The element that's selected by default, as specified by the SELECTED attribute index The index of an option length The number of options in a select object name The name of an options element as specified by the NAME attribute selected A Boolean value specifying the current selection state of an option element selectedIndex The index of the selected option text The text following an <OPTION> tag that's displayed for an option element value The value of the name of an option element as specified by the VALUE attribute

In addition, select objects use the blur() and focus() methods described earlier, and the corresponding onBlur and onFocus event handlers, as well as the onChange event handler.

Submit Object

A submit object is a special button on an HTML form that causes a form's data to be submitted. A submit object supports the name and value properties discussed previously, as well as the click() method and the corresponding onClick event.

Link Object

A link object is some text or image that acts as a hyperlink. When the link object is clicked, the link reference is loaded into its target window.

Note

A document stores link objects in an array that can be referenced by a link element. The length property reflects the number of links in a document.

Link Object Properties

A link object has several properties as listed in Table 9.18.

Table 9.18. Link object properties.

Property Description hash A string beginning with a hash mark (#) that specifies an anchor name in a URL host Specifies the *hostname:port* element of a URL hostname Specifies the host and domain name, or IP address, of a network host href Specifies the entire URL. pathname Specifies the *url-path* portion of the URL port Specifies the communications port that the server uses for communications protocol Specifies the beginning of the URL, including the colon search Specifies a search query target Represents the TARGET attribute (see the section "Anchor Object")

Link Object Event Handlers

Link objects support two event handlers:

- onClick, which occurs in response to the link being clicked.
- onMouseOver, which occurs when the mouse moves over a link.

Anchor Object

An anchor object is some text that can be the target of a link object. Anchor objects use the HTML <A> tag, and

a document stores anchors in an array that can be referenced by an element index. A simple example of an anchor is as follows:

```
<A NAME="SomeAnchorName" SomeAnchorText </A>
```

In this example, the NAME attribute specifies a name called *SomeAnchorName* that can be jumped to from a link within the current document. The text *SomeAnchorText* specifies the text to display at the anchor. To jump to this anchor from somewhere within a document, you can use the following simple link object:

```
<A HREF=SomeAnchorName Go to the SomeAnchorName anchor.</A>
```

Built-In Objects

Built-in JavaScript objects don't have visible user interfaces, and they don't use event handlers. They are used within an HTML page to provide ancillary services for your scripts and are restricted to properties and methods. Three built-in objects are discussed in this chapter:

- Date Object
- String Object
- Math Object

Date Object

The date object represents a date and has several useful methods available for working with dates within a document.

Date Object Methods

The date object methods are listed in Table 9.19.

Table 9.19. Date object methods.

Method Description getDate Returns the day of the month for the specified date (an integer, 1 through 31) getDay Returns the day of the week for the specified date (an integer, 0 through 6) getHours Returns the hour for the specified date (an integer, 0 through 23) getMinutes Returns the minutes in the specified date (an integer, 0

through 59) getMonth Returns the month in the specified date (an integer, 0 through 11) getSeconds Returns the seconds in the specified time (an integer, 0 through 59) getTime Returns the numeric value (in milliseconds since January 1, 1970) corresponding to the time for the specified date getTimezoneOffset Returns the time zone offset (between local and GMT time) in minutes for the current locale getYear Returns the year in the specified date parse Returns the number of milliseconds in a date string (since January 1, 1970) setDate Sets the day of the month for the specified date (an integer, 1 through 31) setHours Sets the hours for a specified date (an integer, 0 through 23) setMinutes Sets the minutes for a specified date (an integer, 0 through 59) setMonth Sets the month for a specified date (an integer, 0 through 11) setSeconds Sets the seconds for a specified time (an integer, 0 through 59) setTime Sets the number of milliseconds since January 1, 1970 in the time for a specified date setYear Sets the year for a specified date toGMTString Converts a date to a string, using Internet GMT conventions toLocaleString Converts a date to a string, using a user's system locale conventions UTC Returns the number of milliseconds in a date object (since January 1, 1970) in Universal Coordinated Time (UTC)

String Object

A string object represents a string of characters. The string object has only one property, length, which returns the number of characters in the string.

String Object Methods

The string object provides a wide variety of methods for use in manipulating strings, and these are listed in Table 9.20.

Table 9.20. String methods.

Method Description anchor() Used with the document write() or writeln() methods to create and display programmatically an anchor in a document big() Causes the specified text to be displayed in a big font, just as it would appear using the HTML <BIG> tag blink() Causes the specified text to blink, just as the HTML <BLINK> tag does bold() Causes a string to be displayed as bold text, just as the HTML <B> tag does charAt() Returns the character at the specified index in the string's array of characters fixed() Causes a string to be displayed with a monospace font, just as the HTML <TT> tag does fontcolor() Causes a string to be displayed in a specified color, just as the HTML <FONT COLOR=*color*> tag does fontsize() Causes a string to be displayed in the specified font size, just as the HTML <FONTSIZE=*size*> tag does indexOf() Returns the index of the first occurrence of a character that matches a specified search string italics() Causes a string to be displayed as italic text, just as the HTML <I> tag does lastIndexOf() Returns the index of the last occurrence of a character that matches a specified search string link() Creates a hypertext link to jump to another URL. small() Causes a string to be displayed with a small font, just as the HTML <SMALL> tag does strike() Causes a string to be displayed as struck-out text, just as the HTML <STRIKE> tag does sub() Causes a string to be displayed as subscript, just as the HTML <SUB> tag does substring() Returns a substring from within a given string object sup() Causes a string to be displayed as superscript, just as the HTML <SUP> tag does toLowerCase() Converts a string to all lowercase letters toUpperCase() Converts a string to all uppercase letters

Math Object

The Math object has built-in properties and methods for mathematical constants and functions. This makes the use of complex mathematical equations easy in JavaScript! The following sections examine the properties and methods provided by this useful object.

Math Object Properties

The math object properties are actually just common, precalculated mathematical constants. These properties are very useful when performing complex calculations and are described in Table 9.21.

Table 9.21. Math object properties.

Property Description E Euler's constant and the base of natural logarithms (about 2.718) LN2 The natural logarithm of two (about 0.693) LN10 The natural logarithm of ten (about 2.302) LOG2E The base 2 logarithm of e (about 1.442) LOG10E The base 10 logarithm of e (about 0.434) PI The ratio of the circumference of a circle to its diameter (about 3.14159) SQRT1_2 The square root of 1/2, or, one over the square root of two (about 0.707) SQRT2 The square root of two (about 1.414)

Math Object Methods

The Math object methods are useful mathematical functions, most with equivalents in the C runtime library. Table 9.22 lists these methods.

Table 9.22. Math object methods.

Method Description abs() Calculates the absolute value of a number acos() Calculates the arc cosine (in radians) of a number asin() Calculates the arc sine (in radians) of a number atan() Calculates the arc tangent (in radians) of a number ceil() Returns the smallest integer greater than or equal to a number cos() Calculates the cosine of a number exp() Returns Euler's constant to the *nn* power, where *nn* is a number supplied as a parameter. floor() Returns the smallest integer less than or equal to a number log() Calculates the natural logarithm (base e) of a number max() Returns the larger of two numbers min() Returns the smaller of two numbers pow() Calculates a base number to the exponent power random() Returns a pseudo-random number between zero and one round() Rounds a number to the nearest integer sin() Calculates the sine of a number sqrt() Calculates the square root of a number tan() Calculates the tangent of a number

---

Note

The random() method is available on UNIX platforms only.

Now that you've looked at the objects, properties, methods, and events offered by JavaScript, take a look at some examples in HTML code.

Creating JavaScript in HTML

A JavaScript applet is created using the HTML <SCRIPT> tag and by specifying the language to be JavaScript. Everything within the script block is part of the JavaScript, but standard HTML comments within a script block can be used to hide the script from browsers that don't support JavaScript. For example, the following HTML code uses a minimal script that displays the string "JavaScript rocks!" on the page box if the browser supports JavaScript:

```
<HTML>

<HEAD>

<TITLE>JavaScript Example 1</TITLE>

</HEAD>

<BODY>

<H1>JavaScript Example 1</H1><HR>

<SCRIPT LANGUAGE="JavaScript">

   document.write("JavaScript rocks!!")

</SCRIPT>

</BODY>

</HTML>
```

Figure 9.1 shows the result of the script in Internet Explorer 3.0.

Figure 9.1. A simple JavaScript in Internet Explorer 3.0.

This is fine for JavaScript-capable browsers, but browsers that don't understand JavaScript will simply display the JavaScript code in the page, as the CompuServe Mosaic browser does in Figure 9.2.

Figure 9.2. A script in a browser that doesn't understand JavaScript.

Because some users will undoubtedly use browsers that don't support JavaScript, you can prevent the script from

appearing at all in these browsers by enclosing the entire script code in HTML comments. The revised code from the preceding example would give you this:

```
<HTML>

<HEAD>

<TITLE>JavaScript Example 1</TITLE>

</HEAD>

<BODY>

<H1>JavaScript Example 1</H1><HR>

<SCRIPT LANGUAGE="JavaScript">

<!-- Hide JavaScript from old browsers

   document.write("JavaScript rocks!!")

// End code hiding -->

</SCRIPT>

</BODY>

</HTML>
```

JavaScript Functions

The real power of JavaScript begins to show when you add functions to the mix. As any modern programming language should, JavaScript supports the notion of functions that live in one area and can be called from elsewhere in the code. Where do JavaScript functions live in HTML? Typically in the <HEAD> section. This is because the function code will be loaded and interpreted before the page is displayed, and any JavaScript source that calls a function will have it available instantly.

A JavaScript function is defined (appropriately) by the keyword function, and it must fall within a <SCRIPT> block. The function is given a name, and this name can be called from a JavaScript code. Take a look at a function called displayMessage(), which takes a typeless parameter called msg and sends it to the JavaScript alert() method.

```
function displayMessage(msg)
```

```
{

   alert(msg)

}
```

This function could then be called by placing a form pushbutton on the page and calling the dislayMessage() function in response to the button's onClick event, like this:

```
<FORM>

   <INPUT TYPE="button"

      VALUE="Click Here..."

      onClick="displayMessage('Test message...')">

</FORM>
```

Note

The string Test message... is surrounded with single quotes. This is because they are nested within the double quotes used for the displayMessage() function call by the onClick event.

The complete source code for this function example is given in Listing 9.1.

Listing 9.1. Calling a function from within a JavaScript.

```
<HTML>

<HEAD>

<TITLE>JavaScript Example 2</TITLE>

<SCRIPT LANGUAGE="JavaScript">

<!--

function displayMessage(msg)

{

   alert(msg)
```

```
}

-->

</SCRIPT>

</HEAD>

<BODY>

<H1>JavaScript Example 2</H1><HR>

This script calls a simple function...click the button!

<FORM>

   <INPUT TYPE="button"

      VALUE="Click Here..."

      onClick="displayMessage('Test message...')">

</FORM>

</BODY>

</HTML>
```

In this example, the string "Test message..." is sent to the function displayMessage(), causing an alert box to appear. Now extend this into a more useful example by enabling a user to specify the text displayed in the alert box.

For this example, a text object named text1 is used to enable the user to input the text to display in the alert box. The showMessage() function is modified here to take a form object as a parameter, like this:

```
<SCRIPT LANGUAGE="JavaScript">

<!-- Updated showMessage() function takes a form as a parameter

function showMessage(form)

{

   if (form.text1.value.length > 0)

      alert("You entered the text: " + form.text1.value)

   else
```

```
      alert("You must enter some text!")

}

-->

</SCRIPT>
```

Notice that if the length of the text isn't greater than zero, the script tells the user to enter some text; otherwise, the text is displayed using the form.text1.value property to get the string.

To enhance the page even more, the example uses a date object to extract information about when the page was last modified. This information is then written to the bottom of the page to provide automatic update information whenever the page changes, like this:

```
<H5>

<SCRIPT LANGUAGE="JavaScript">

<!-- Hide script from old browsers

   update = new Date(document.lastModified)

   month  = update.getMonth() + 1

   date   = update.getDate()

   year   = update.getYear()

   document.writeln("<I>Last modified: " +

      month + "/" + date + "/" + year + "</I>")

// End script hiding -->

</SCRIPT>

</H5>
```

The complete code listing for this example is given in Listing 9.2.

Listing 9.2. A page that gathers text data from a simple form and displays the document's last modified date.

```
<HTML>

<HEAD>
```

```
<TITLE>JavaScript Example 3</TITLE>
<SCRIPT LANGUAGE="JavaScript">
<!-- Updated showMessage() function takes a form as a parameter
function showMessage(form)
{
   if (form.text1.value.length > 0)
      alert("You entered the text: " + form.text1.value)
   else
      alert("You must enter some text!")
}
-->
</SCRIPT>
</HEAD>
<BODY>
<H1>JavaScript Example 3</H1>
<HR>
This script uses the data you enter in the text area and
 displays it in an alert box...click the button!
<!-- Start a form for getting the text to use -->
<FORM>
   <INPUT TYPE="button" VALUE="Click Here..."
      onClick="showMessage(this.form)">
   <INPUT TYPE="text" SIZE=40 MAXLENGTH=256 NAME="text1"></p>
</FORM>
<!-- Show the last modified date in the document with this simple script -->
```

```
<H5>

<SCRIPT LANGUAGE="JavaScript">

<!-- Hide script from old browsers

   update = new Date(document.lastModified)

   month  = update.getMonth() + 1

   date   = update.getDate()

   year   = update.getYear()

   document.writeln("<I>Last modified: " +

      month + "/" + date + "/" + year + "</I>")

// End script hiding -->

</SCRIPT>

</H5>

</BODY>

</HTML>
```

The results of this example can be seen in Figure 9.3, with the text box value displayed in an alert box, and the date the document was last modified displayed as part of the document.

Figure 9.3. EX3.HTM as it appears in Internet Explorer 3.0.

These few small examples should give you an idea of the power and ease of JavaScript. In Chapter 12, "Advanced Web Page Creation," you'll see some more useful and exciting examples for working with JavaScript and frames.

Summary

JavaScript provides a powerful way for Web page authors to add interactivity to Web pages with a well-structured scripting language. Programmers familiar with C, C++, or Java should have no trouble learning it. When combined with other tools and some imagination, JavaScript can provide the means to create a truly exciting Web browsing experience for the user.

The next chapter examines Microsoft's Web authoring tool FrontPage, a versatile and all-around excellent package for designing and maintaining Web sites.


TOC

Chapter 10

Using Microsoft FrontPage

by Rob McGregor

Creating and maintaining Web sites used to be a tedious and mind-numbing process, especially when the sites grew large and unwieldy. But Microsoft's FrontPage is changing all that. Although the version 1.1 release isn't yet a complete Web site solution, it's very close. By using its excellent Web creation and maintenance tools, you can be a Web master in no time.

What Is FrontPage?

Microsoft FrontPage was created to assist you in the design and management of Web sites. FrontPage isn't just about creating personal Web sites, though; it's about creating exciting, interactive sites that can represent your company. FrontPage helps take the tedium out of managing your Web site, and it saves you a lot of time in the initial development process. FrontPage enables you to spend your time being creative by providing you with a WYSIWYG HTML editor and a visual, hierarchical representation of your entire Web site (or in FrontPage terms: your "web").

Adding new pages to your web is as simple as a few clicks of the mouse. FrontPage automatically links new pages to existing pages, and it provides you with a plethora of templates and wizards so that you don't have to start from scratch (another valuable time saver). Even if you're a diehard HTML author whose favorite editor is Notepad, you'll see just how much time and effort you can save after just a few minutes using the FrontPage Web Editor.

The FrontPage Explorer, Editor, and Personal Web Server

FrontPage combines the power of a Web explorer, a WYSIWYG HTML editor, and a personal Web server so that you can whip out a professional-looking site in no time. Take a look at these amazing tools, starting with the FrontPage Explorer.

The FrontPage Explorer

The FrontPage Explorer is a Web site management utility that is one of FrontPage's major assets. Using this tool, you can maintain an incredibly complex Web site easily. Figure 10.1 shows the main window of the FrontPage Explorer, with an outline view in the left pane and a graphical hierarchy of Web site links in the right pane. As items in the outline view are selected, the corresponding site links are displayed in the link view. The FrontPage Explorer makes locating pages and keeping track of documents and links incredibly easy.

Note

The right pane can also display a Summary View, a detailed listing of all files used in a web. Access the Summary View by choosing the View, Summary View menu command.

Figure 10.1. The FrontPage Explorer.

By clicking through the various items in the outline view, you can see the relationships and links among the documents that make up a site.

Consistent, Automatic Styles

Setting or changing an overall style for a Web site is quick and easy with FrontPage. You can easily edit the colors, navigation links, and company logo for each page in a site from a single location.

The tree view in the outline pane shows the internal links used by each page for these common items. For instance, instead of using a company logo image on each page that needs it, the image is simply referenced from a logo page. Changing the logo on that one page updates the image throughout the Web site, wherever the logo image is referenced. Similarly, colors and navigation links are each referenced by their respective master pages. This makes setting or changing an overall Web site style quick and easy.

Double-clicking a page icon in the link view launches the FrontPage Editor to allow visual editing of the source file for the selected page. (See the section "The FrontPage Editor.")

Managing Your Web

The ability to effectively manage a large Web site is crucial, and the FrontPage Explorer's menus offer many useful commands for performing Web creation and maintenance. This section examines these menus to get an overview of the tools the FrontPage Explorer provides.

The File Menu

The File menu provides commands that you use to manage entire Web sites. Table 10.1 describes the available commands.

Table 10.1. FrontPage Explorer File menu commands.

Menu Command Enables You To New Web Create a new web from existing templates and wizards Open Web Open an existing web Close Web Close a web currently open in the FrontPage Explorer Copy Web Copy an existing web to a new location Delete Web Delete an existing web Import Import files into an existing web Export Selected Export the currently selected image or HTML file from the web to a specified location

The Edit Menu

The Edit menu provides commands to edit Web files, pages, images, documents, tasks, and properties. Table 10.2 describes the available commands.

Table 10.2. FrontPage Explorer Edit menu commands.

Menu Command Enables You To Delete Delete the currently selected page or file from a Web. Add To Do Task Add a task to the current web's To Do list, linking the task to the active page. Open Open a selected page or file in either the outline view or the link view. This is the same result as double-clicking an item in the FrontPage Explorer. Open With Open the currently selected file with any editor associated with various file types (by using the Tools|Configure Editors menu command). Properties View and edit the properties of the currently selected page, such as: title and URL of the currently selected page, author, comments, and notes.

The View Menu

The View menu provides commands to control user-interface elements like toolbars. Table 10.3 describes the available commands.

Table 10.3. FrontPage Explorer View menu commands.

Menu Command Enables You To Toolbar Display or hide the toolbar Status Bar Display or hide the status bar Split Move the vertical bar that divides the FrontPage Explorer's left and right panes Link View View a web as a graphical hierarchy of links Summary View View a web as a detailed list of files used in the web Links to Images Display or hide all links to images in a web Repeated Links Display or hide multiple links from one page to another page Links Inside Page Display or hide any links within a page targeting anchors that lie within that same page Refresh Update all views within the FrontPage Explorer

The Tools Menu

The Tools menu provides commands to configure settings for various web elements. Table 10.4 describes the available commands.

Table 10.4. FrontPage Explorer Tools menu commands.

Menu Command Enables You To Web Settings Get information about the current web or set advanced features for the web. Permissions Set permissions for administrators, authors, and end users of a web. Administrators have full reign over a web, its pages, and security. Authors can only create and delete pages. End users can only read pages, or they can be denied access to a web either partially or altogether. Configure Editors Associate an editor with a file type. Change Password Change a password. Proxies Register a proxy server name, if used by your local network, and register all hosts located within your firewall. Verify Links Verify internal and external links in a web and repair broken or dubious links. Recalculate Links Update the search text index (created by a Search bot) and regenerate dependencies. Stop Cancel a request that has been sent to a server. Show FrontPage Editor Launch or display the FrontPage Editor. Show To Do List Display the To Do List for the current web.

That completes this overview of the FrontPage Explorer, a sophisticated and powerful Web site management tool. Next up is the FrontPage Editor.

The FrontPage Editor

The FrontPage Editor is a full-featured HTML word processor and page layout program that gives you full WYSIWYG editing of HTML documents, as you can see in Figure 10.2. The FrontPage Editor provides you with a clean environment where you can create professional Web pages in a fraction of the time it would take using a lesser editor (like Notepad).

Figure 10.2. The FrontPage Editor gives you full WYSIWYG editing of HTML documents.

Note

The FrontPage Editor uses ActiveX objects called *bots* to enable dynamic evaluation and execution of data that is typically saved as HTML when a page is saved to the server.

Editing Your HTML Files

The FrontPage Editor provides an abundance of tools for creating advanced HTML files, with fancy layouts, difficult tables, and easy frames. Take a look at the menu commands the Editor provides.

The File Menu

The File menu provides commands that enable you to work with files and to set page properties and print options. Table 10.5 describes the available commands.

Table 10.5. FrontPage Editor File menu commands.

Menu Command Enables You to New Create a new page using an existing page template or page wizard in the New Page dialog box Open File Open a text (TXT), rich text format (RTF), or HTML (HTM or HTML) file for editing Open from Web Open a page from the current web Open Location Open a page or other file, directly from the World Wide Web, for editing Close Close the active page Save Save the active page Save As Save the active page to a new page in the current web, or to a file, or create a page template from the active page Save All Save all open pages in HTML format simultaneously Page Properties Set properties for the appearance of the active page, edit the page title, and assign a base URL Page Setup Set header, footer, and margin information for the active page Print Preview Preview the active page onscreen as it would appear when printed Print Print the active page Exit Close the FrontPage Editor

Setting page properties is easy with the File|Page Properties menu command, which calls up the dialog box shown in Figure 10.3.

Figure 10.3. The Page Properties dialog box.

The Edit Menu

The Edit menu provides the standard Edit menu commands typical of a word processor, plus the specialized commands described in Table 10.6.

Table 10.6. FrontPage Editor Edit menu commands.

Menu Command Enables You To Add To Do Task Add a task to the current web's To Do list, linking the task to the active page Bookmark Create a bookmark (a link target or anchor) for any currently selected text Link Create or modify a link, using selected text, to a URL in the current web or in the World Wide Web Unlink Remove a link from selected text without deleting the text characters Properties Open a Properties dialog box for a selected object, including bookmarks, bots, characters, form fields, entire forms, horizontal lines, hotspots, images, links, paragraphs, table cells, and entire tables

The View Menu

The View menu provides commands to display or hide the various toolbars used by the Editor and to display or hide the status bar. The View menu also enables you to display or hide format marks. These are normally invisible page elements like hard returns, bookmarks, and form outlines.

The Insert Menu

The Insert menu provides commands to insert various standard HTML tags into a page at the insertion point specified by the current cursor location. Table 10.7 describes the available commands.

Table 10.7. FrontPage Editor Insert menu commands.

Menu Command Enables You To Heading Insert any of six heading paragraph sizes, from <H1> to <H6>. List Insert any of four types of lists: bulleted, numbered, directory, or menu. Definition Insert any of three types of definition formatting: list, term, and definition. Form Field Insert named form fields with parameters set by dialog boxes. The available fields are single-line text box, multiple-line text box, check box, radio button, drop-down menu, pushbutton, and image. Paragraph Insert three types of paragraph tags: normal, formatted, and address. Horizontal Line Insert a horizontal line tag <HR>. Line Break Insert one of four types of line breaks: standard line break, clear left margin, clear right margin, and clear both margins. Image Insert an image file in any of the following formats: GIF, JPEG, BMP (Windows and OS/2), TIFF, MAC, MSP, PCD, RAS, WPG, EPS, PCX, and WMF. Bot Insert any of the following types of bots: Annotation bot, Confirmation Field bot, HTML Markup bot, Include bot, Scheduled Image bot, Scheduled Include bot, Search bot, Substitution bot, Table of Contents bot, Timestamp bot. Special Character Insert any of the special characters shown in Figure 10.4 into a page at the insertion point. File Insert a copy of a specified RTF, HTML, or text file at the insertion point.

Figure 10.4. Special characters available for insertion with the Insert|Special Characters menu.

The Format Menu

The Format menu provides commands to select various formatting options for selected characters or paragraphs or to remove formatting from selected characters.

The Format|Character menu command invokes the dialog box shown in Figure 10.5, which lets you choose character styles, font size, color, and position.

Figure 10.5. The Character Styles dialog box.

The Format|Paragraph menu command enables you to choose any available formatting styles from a list

box. These options are also available on the Format toolbar.

The Format|Remove Formatting menu command removes formatting from any selected characters, returning character formatting to the default style.

The Tools Menu

The Tools menu provides a spell-checker, as well as standard browser commands for navigating URLs, bookmarks, and so on. The available browser commands are as follows:

- Forward
- Back
- Follow Link
- Reload
- Stop

You can also call up the To Do List dialog box and FrontPage Explorer from the Tools menu.

The Table Menu

The Table menu provides commands to create and edit HTML tables, and Table 10.8 describes the available menu commands.

Table 10.8. FrontPage Editor Table menu commands.

Menu Command Enables You To Insert Table Insert a table designed to your specifications with the Insert Table dialog box. (See Figure 10.6.) Insert Rows or Columns Insert a specified number of rows or columns at the insertion point. Insert Cell Insert a new cell at the insertion point. Insert Caption Insert a caption above a table. Merge Cells Merge selected cells into a single cell. Split Cells Split selected cells into a specified number of cells. Select Cell Select the cell at the insertion point. Select Row Select an entire row of cells. Select Column Select an entire column of cells. Select Table Select an entire table. Table Properties Adjust table properties with a trimmed-down version of the Insert Table dialog box. (See Figure 10.6.)

Figure 10.6. The Insert Table dialog box.

Some Caveats

The FrontPage Editor is great for developing frames and tables quickly and almost effortlessly, but be careful about editing existing HTML files containing Java applets, JavaScript, or VBScript. The Editor has a tendency to replace certain important scripting characters with equivalent HTML symbols.

For example, the double-quotes (") in the following JavaScript expression are replaced with their HTML equivalent ("):

```
document.write ("Howdy.")
```

That gives you this unusable code:

```
document.write (&quot;Howdy.&quot;)
```

The Editor also has a tendency to "scrunch up" nicely formatted code into a mass of illegible code. For example, the nicely formatted code in Listing 10.1 becomes the mangled code in Listing 10.2.

Caution

Don't retire Notepad just yet. Version 1.1 of the FrontPage Editor doesn't support Netscape bullet styles, Java applets, JavaScript, or VBScript directly. In fact, the editor is oblivious to scripting and can change existing script code such that it won't run without editing by hand. You'll probably still need your trusty Notepad to do some final tweaking.

Listing 10.1. An HTML file as it appears before importing into the FrontPage Editor.

```
<html>

<head>

<title>My Title</title>

</head>

<body>

<h2>

Sample Scruncher Page
```

```
</h2>

<hr>

<h3>This page is formatted with lots of white space: </h3><p>

 <li> White space lets us understand the code better.<br>

 <li> Code with lots of white space is aesthetically pleasing.<p>

For this example page, the following Java applet should be spaced

 out with one item per line:

<p>

<applet

   codebase = "../java/SomeApplet"

   code     = SomeApplet.class

   id       = SomeApplet

   width    = 160

   height   = 120>

   <param name = "Param1"  value = "10">

</applet>

<hr>

<a href="SomeApplet.java"><b>The source</b></a>

</body>

</html>
```

Listing 10.2. The HTML file from Listing 10.1 as it appears after importing into the FrontPage Editor and saving.

```
<.DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
```

```
<html>

<head>

<title>My Title</title>

<meta name="GENERATOR" content="Microsoft FrontPage 1.1">

</head>

<body>

<h2>Sample Scruncher Page </h2>

<hr>

<h3>This page is formatted with lots of white space: </h3>

<p>White space lets us understand the code better.<br>

Code with lots of white space is aesthetically pleasing.</p>

<p>For this example page, the following Java applet should be

spaced out with one item per line: </p>

<p><applet codebase="../java/SomeApplet" code="SomeApplet.class"

id="SomeApplet" width="160" height="120"> <param name="Param1"

value="10"> </applet> </p>

<hr>

<p><a href="SomeApplet.java"><b>The source</b></a> </p>

</body>

</html>
```

All these little idiosyncrasies aside, this really is one of the best HTML editors on the market today. The revisions for the upcoming release of FrontPage 97 (slated to be included with Microsoft Office 97) should fix these little problems and add a lot of ActiveX scripting and control support. Getting used to FrontPage now will put you that much further ahead of the pack later. Next, let's take a quick look at the FrontPage Personal Web Server.

The FrontPage Personal Web Server

The FrontPage Personal Web Server doesn't offer a lot of interactivity, just a File|Exit menu command and a Help|About menu command. Although it may not look like much, as you can see in Figure 10.7, there's a lot of power lurking behind its minimal user interface.

Figure 10.7. The Personal Web Server.

The Personal Web Server is really just that, a personal Web server. The FrontPage Explorer expects to be connected to a Web server when creating and editing webs, and the Personal Web Server provides the connectivity you need to simulate Web server functions when you're not connected to a real server.

The Personal Web Server was designed to allow site builders to test their Web sites thoroughly before putting the site online. But don't get the wrong impression. The Personal Web Server has the capability to actually function as a real, if somewhat limited, Web server for the World Wide Web. By simply connecting a desktop PC to the outside world with a modem, you can run your FrontPage webs right from a stand-alone machine. Of course, there's a catch. You'll need to obtain a permanent Internet Protocol (IP) address by registering your site. Then the entire world can find and browse your site. Your Web server can be configured using the FrontPage Server Administrator program that ships with FrontPage. (See Figure 10.8.)

Note

If you use a dedicated server machine running a network operating system (NOS), such as Windows NT Server, use the FrontPage Server Administrator in conjunction with the NOS Web server configuration utilities supplied with that machine.

Figure 10.8. The FrontPage Server Administrator controls server configurations.

Web Templates and Wizards

FrontPage comes equipped with web templates and web wizards suitable for many common types of Web sites. The templates simply provide a predefined layout for a page. The wizards actually walk you, step by step, through the process of creating a complete Web site (with all of the trimmings).

Using a FrontPage Wizard

Wizards are a big help in breaking down complex tasks into simple steps, and the FrontPage wizards do a tremendous job. The most astounding and comprehensive of the wizards included with FrontPage is the Corporate Presence Web Wizard. This excellent tool walks you through the many steps needed to create a basic corporate Web site, complete with navigation tools and example text.

The Corporate Presence Web Wizard contains several pages that you must step through, supplying the appropriate information each step of the way. Figure 10.9 shows the first step, where you decide which pages to include in your web.

Figure 10.9. The first step of the Corporate Presence Web Wizard.

You must also decide which items to include on your corporate home page as seen in Figure 10.10. This gives you a good starting point for your new Web site.

Figure 10.10. Deciding which items to include on your corporate home page.

Each page selected for the site in the first step has its own choice of content selections. Figure 10.11 shows the options for the What's New Page.

Figure 10.11. Choosing What's New items.

The Feedback Form provides a means of getting feedback from people that visit your site, and the options in the wizard pages shown in Figured 10.12 through 10.15 make it easy to set up the information you'd like to get from your visitors.

Figure 10.12. Deciding which feedback items to request from users of your web.

Figure 10.13. Choosing items common to each Web page, such as company logo, e-mail address, and copyright information.

Figure 10.14. Choosing a graphics style for the Web site, such as conservative, flashy, or cool.

Figure 10.15. Picking custom or default link colors and background for the Web site.

Providing users of your site with quick access to contact information is another important item. Setting up contact information is done through the wizard's contact page, as seen in Figure 10.16. Here you can specify phone and FAX numbers, as well as e-mail addresses.

Figure 10.16. Specifying important company contact information.

After all of the wizard's steps are completed, the FrontPage Explorer creates a new corporate web for you, generating all the starter files you selected and filling in the styles and information you specified in the wizard. Once the web is ready, a To Do List is displayed, as shown in Figure 10.17. The To Do List makes it easy to finish up your new web's look by replacing the generic logo with your company logo and inserting the proper text and graphics as needed.

Figure 10.17. The To Do List reminds you of tasks yet to be completed.

Once your To Do List is completed, your Web site should be ready for a test drive, some customization, and finishing touches.

Note

In Chapter 14, "Advanced Web Page Creation," you'll see how to achieve a small, complete Web site that's ready for prime time.

FrontPage Tables

FrontPage makes quick work of tables. Tables are easily created and modified using the Table menu commands discussed earlier, in the section "The FrontPage Editor." This section reviews the steps needed to create and modify a table.

Creating and Modifying a Table

To begin, open the FrontPage Editor and select the File|New menu command, and accept the default "Normal Page" template from the dialog box that appears. Next, select the Table|Insert Table menu command. Set the Insert Table controls to the values in Table 10.9.

Table 10.9. Values for the Insert Table dialog box controls.

Control Value Rows 5 Columns 4 Alignment Center Border Size 1 Cell Padding 1 Cell Spacing 1 Specify Width 75%

After you choose the OK button, your page should look like Figure 10.18. Now add a caption for the table by selecting the Table|Insert Caption menu command and then type the following caption: A FrontPage Table.

Figure 10.18. The new page with the empty, raised border table.

Next, format the caption by selecting the entire text and clicking the Bold and Italic buttons on the Format toolbar (the toolbar's tooltips will reveal which tool buttons these are). Follow this by clicking the Increase Text Size Formatting toolbar button twice. The caption should now look like the one in Figure 10.19.

Figure 10.19. The table with a caption.

Click the left-middle cell to position the insertion point and choose the Table|Select Row menu command, followed immediately by the Table|Merge Cells menu command. This merges the middle row of cells into a single cell as you can see in Figure 10.20. It couldn't be much easier.

Figure 10.20. The modified table with a single cell in the third row.

Filling the Table

After adding some text and a few images to the table, you should have a good feel for creating and modifying tables. To add text to a cell, simply click the cell and type. The WYSIWYG Editor makes it easy. You can add images to cells by selecting the Insert|Image command and choosing the desired image from a URL or a file. After the text and image are in place, select the entire table and click the Center tool button on the Format toolbar. Depending on the text and image inserted into the cells, you get a table similar to the one shown in Figure 10.21.

Figure 10.21. The Finished table with text and an image in the cells.

The HTML Source for the Table

To appreciate what FrontPage is doing behind the scenes, take a look at Listing 10.3, which gives the source code for the completed page.

Listing 10.3. The HTML source for the table.

```
<.DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">

<html>

<head>

<title>Untitled Normal Page</title>

<meta name="GENERATOR" content="Microsoft FrontPage 1.1">

</head>

<body>

<div align=center><center>

<table border=1 width=75%>

<caption align=top><font size=5><em><strong>A FrontPage Table
```

```
</strong></em></font></caption>
<tr><td width=25%><p align=center><font size=5><strong>
<tt>Putting</tt></strong></font></p>
</td><td width=25%><p align=center><font size=5><strong>
<tt>Text</tt></strong></font></p>
</td><td width=25%><p align=center><font size=5><strong>
<tt>In</tt></strong></font></p>
</td><td width=25%><p align=center><font size=5><strong>
<tt>Cells</tt></strong></font></p>
</td></tr>
<tr><td width=25%><p align=center><font size=5><strong>
<tt>Is</tt></strong></font></p>
</td><td width=25%><p align=center><font size=5><strong>
<tt>An</tt></strong></font></p>
</td><td width=25%><p align=center><font size=5><strong>
<tt>Easy</tt></strong></font></p>
</td><td width=25%><p align=center><font size=5><strong>
<tt>Task.</tt></strong></font></p>
</td></tr>
<tr><td colspan=4 width=100%><p align=center>
<img src="fountain.gif" align=bottom width=560 height=225></p>
</td></tr>
<tr><td width=25%><p align=center><font size=5><strong>
```

```
<tt>Putting</tt></strong></font></p>
</td><td width=25%><p align=center><font size=5><strong>
<tt>Images</tt></strong></font></p>
</td><td width=25%><p align=center><font size=5><strong>
<tt>In</tt></strong></font></p>
</td><td width=25%><p align=center><font size=5><strong>
<tt>Cells</tt></strong></font></p>
</td></tr>
<tr><td width=25%><p align=center><font size=5><strong>
<tt>Is</tt></strong></font></p>
</td><td width=25%><p align=center><font size=5><strong>
<tt>Also</tt></strong></font></p>
</td><td width=25%><p align=center><font size=5><strong>
<tt>Very</tt></strong></font></p>
</td><td width=25%><p align=center><font size=5><strong>
<tt>Easy.</tt></strong></font></p>
</td></tr>
</table>
</center></div>
</body>
</html>
```

Summary

Overall, FrontPage is an excellent Web site creation, administration, and maintenance package that provides you with all the tools you need for a complete, professional Web site. The combination of the FrontPage Explorer, the FrontPage Editor, the FrontPage Personal Web Server, and the FrontPage Server Administrator all perform quite well and produce great results.

Although the FrontPage WebBots give you some degree of interactivity, many additional interactive elements can be added to further enhance your site once pages are laid out and finished in FrontPage. To this end, the next chapter turns your attention to adding ActiveX controls and scripts to Web pages with the ActiveX Control Pad. In Chapter 14, you'll learn to develop a complete set of pages that utilize many of the tools discussed in this book to create a truly interactive and exciting (if small) Web site.

Chapter 11

Using ActiveX Control Pad

The World Wide Web has made a dramatic impact on the computing paradigm in the last couple of years. It has changed the way people access information and communicate with each other. This is largely due to the development of technologies that enable the Web to interact with the user, leading to a more meaningful and fulfilling Web surfing experience.

Leading the way toward making the Web a powerful medium for accessing information, experiencing multimedia content, and enabling integration of the Web with the desktop is Microsoft's ActiveX technology. The core component of the ActiveX set of technologies is ActiveX Controls, which are used to author Web pages with *active* content.

The Microsoft Internet Explorer 3.0 Web browser and the Mosaic Web browser support Web pages with active content developed using ActiveX Controls. NCompass Labs, Inc. have developed an add-on called ScriptActive to the Netscape Navigator Web browser that adds ActiveX and VBScript support to the Navigator Web browser. More information about ScriptActive can be found at the NCompass Web site at http://www.ncompasslabs.com.

Microsoft Internet Explorer 3.0 will be available for the Macintosh and the various UNIX platforms soon, making ActiveX technology truly universal in appeal.

Web Pages with ActiveX Controls

ActiveX controls are software components that can be used to develop interactive, advanced Web pages that incorporate multimedia support, interface with back-end databases, and receive input from the user. The process of authoring ActiveX Web pages begins with the third-party developers who develop ActiveX Controls for use by Web designers and Web authors. These controls are distributed with a developer's license to Web site developers, who then incorporate the functionality provided by the controls into their Web pages.

ActiveX controls are source-code independent, and they can work with other controls developed in different languages. It is the Webmaster who orchestrates the interaction of the controls with each other and with the user.

ActiveX controls are binary objects having properties, events, and methods:

- Properties define the appearance or behavior of the control. They can be set at design time and can also be changed at runtime.
- Events are triggered during runtime. An event may be generated in response to user input or other criteria such as time elapsed, and so on. The events are notified to the browser, which can then respond to the event. Scripts that are invoked in response to specific events are called "event handlers."
- Methods incorporate the functionality of the ActiveX control. Methods are invoked by the browser, and the ActiveX control performs the appropriate function as required.

The communication between the browser and the ActiveX control is achieved through the use of snippets of code in a scripting language such as Visual Basic Scripting Edition (VBScript) or JavaScript. If you need more information, refer to Chapters 8, "VBScript" and 9, "JavaScript," of this book, which discuss these scripting languages.

Embedding ActiveX Controls in Web Pages

Web pages incorporate ActiveX controls using the HTML <OBJECT> tag. The syntax for the <OBJECT> tag when used for embedding ActiveX controls is as follows:

```
<OBJECT
CLASSID="CLSID:class-identifier"
CODEBASE="URL where control can be downloaded from"
DATA="url to control's object date"
ID=control_name
ALIGN=TEXTTOP | MIDDLE | TEXTMIDDLE | BASELINE | TEXTBOTTOM | LEFT | CENTER | RIGHT
WIDTH=width of control
HEIGHT=height of control
BORDER=TRUE | FALSE
>
<PARAM NAME="parameter name" VALUE="parameter value">
.
.  (list of name-value pairs for controls having multiple parameters)
.
<PARAM NAME="parameter name" VALUE="parameter value">
</OBJECT>
```

The class-identifier is a unique 128-bit value that identifies each ActiveX control uniquely. This 128-bit value is stored in the registry on the system on which the ActiveX control was installed and registered. The Web developer has to specify this class-identifier in the HTML syntax for incorporating the control in the HTML document.

Interactive Web Pages: Overview of Scripting

Events that are generated by ActiveX controls are handled by procedures of code that are embedded in the HTML file in the form of a script. The two popular scripting languages are Visual Basic Scripting Edition (VBScript) and JavaScript.

The HTML syntax for embedding scripts in Web pages is

```
<SCRIPT LANGUAGE = "VBScript">
```

```
<!--

Sub SomeSubroutine(SomeParameter)

.

.

.

End Sub

-->

</SCRIPT>
```

Refer to Part 2, "ActiveX Scripting," of this book for more information on writing scripts.

ActiveX Control Pad

Using ActiveX Controls in Web authoring involves carefully noting the class-identifier for each ActiveX Control, specifying the property values for the control by enlisting them in the <PARAM> and <VALUE> tag pairs, and writing the necessary scripts to achieve the required interactivity. This process becomes extremely unwieldy after a while, especially if you are working with a considerable number of controls.

Another obstacle that Web designers are confronted with is that they cannot specify precise positioning information for various ActiveX controls on a Web page using pure HTML 3.2. HTML 3.2 does not support specifying coordinates for placing objects on Web pages.

Microsoft, sticking to its commitment of making life easier for developers, has come to the rescue. Recognizing the need for a development environment for authoring HTML pages using ActiveX Controls, it has developed the first version of what is called the ActiveX Control Pad.

The ActiveX Control Pad provides an integrated development environment for authoring Web pages with active content. The ActiveX Control Pad version 1.0 has the following features:

- Text Editor: Used for simple editing of HTML pages. It doesn't have special HTML editing features, and it is a simple text editor.
- Object Editor: With the Object Editor you can insert ActiveX Controls in the HTML file open in the Text Editor. The Object Editor has a Properties window, which provides a visual representation of the properties of the ActiveX control so that you can edit the properties easily.
- Script Wizard: The Script Wizard makes adding simple scripts to HTML files a snap. Using its powerful intuitive interface, you can easily specify the actions to be taken in response to specific events, and the Script Wizard takes care of writing the script for you. The Script Wizard can be configured to generate either VBScript or JavaScript.
- 2D-Style Page Editor: A WYSIWYG page editor for creating special 2D layout regions within an HTML page. The 2D-style region provides precise placement control of the controls used on the HTML page.
- ActiveX Controls: The ActiveX Control Pad comes with an assortment of ActiveX Controls free for incorporating in your own ActiveX Web pages.

This chapter takes a tour of the ActiveX Control Pad and shows you how it makes authoring Web pages with active content easier than ever.

Note

ActiveX Control Pad does not support authoring HTML pages with Java applets in version 1.0. Microsoft will incorporate support for Java applets in a future version of ActiveX Control Pad. You can expect the newer version with support for Java applets soon after Visual J++ is released.

Getting Started

Let's begin by looking at the basic features of ActiveX Control Pad. We will familiarize ourselves with the ActiveX Control Pad environment first and proceed to developing a simple HTML page with an ActiveX control.

Installing ActiveX Control Pad

The ActiveX Control Pad is available on the accompanying CD-ROM. There are different versions for Windows 95 and Windows NT 4.0, so be careful to use the appropriate version. For using the ActiveX Control Pad, you must have the Microsoft Internet Explorer 3.0 browser installed on your system. If you do not have Internet Explorer 3.0 installed, the Setup will not install the ActiveX Control Pad. If you have not already done so, you can install Microsoft Internet Explorer 3.0 from the accompanying CD-ROM.

Double-click SETUPPAD.EXE to start the setup and installation program. The installation routine will check for the presence of Internet Explorer 3.0 and then guide you through the installation process. The ActiveX Control Pad is installed by default in the C:\PROGRAM FILES\ActiveX Control Pad folder. The ActiveX Control Pad will copy the required program files and online help, and it will register the various ActiveX Controls that are bundled with it on your system.

Launching the ActiveX Control Pad

You can start the ActiveX Control Pad from the Start|Programs|Microsoft ActiveX Control Pad menu. The Control Pad opens up with a blank new HTML file as shown in Figure 11.1.

Figure 11.1. ActiveX Control Pad with blank new HTML file.

You are now ready to begin using the Control Pad to author ActiveX Web pages.

Tip

You can right-click on any HTML file in the Windows Explorer or Folder view to edit it with ActiveX Control Pad. ActiveX Control Pad starts with the HTML file loaded for editing.

The Control Pad is an Multiple Document Interface (MDI) application, so you can edit multiple HTML files at the same time. This means you can cut, copy, and paste HTML tags, objects, and 2D layout regions across multiple files and edit them at the same time.

The Text Editor

The Text Editor has a vertical pane on the left of the HTML document where the ActiveX Control Pad places icons for easy access to ActiveX Controls and 2D layout regions that are embedded in the HTML file.

Tip

Create and edit your HTML file using an HTML editor such as Internet Assistant for Microsoft Word or Microsoft FrontPage. After you have your HTML tags in place, you can proceed to add active content using ActiveX Control Pad.

Adding ActiveX Controls

Place the cursor in the HTML file where you want to insert the ActiveX Control. Select Insert ActiveX Control on the Edit menu. The Control Pad shows a list box displaying all the controls registered on your system.

Table 11.1 lists the controls supplied with the ActiveX Control Pad. The examples in this chapter use several of the controls listed. It is easy to use any ActiveX control once you are familiar with a few of them. You can experiment with the Hot Spot and Web Browser controls yourself, after you are finished reading this chapter.

Table 11.1. ActiveX controls included with ActiveX Control Pad.

Control Description Microsoft ActiveX Image Control 1.0 Display images in common graphics file formats, including BMP, JPEG, and GIF Microsoft ActiveX Hot Spot Control 1.0 Create regions within an Image to respond in different ways Microsoft Forms 2.0 CommandButton Push-button control Microsoft Forms 2.0 CheckBox Check an option Microsoft Forms 2.0 ComboBox Select from drop-down list of options Microsoft Forms 2.0 Frame Group related controls together Microsoft Forms 2.0 Image Display , crop, size images Microsoft Forms 2.0 Label Labels for text and icons Microsoft Forms 2.0 ListBox Select from a scrollable list of options Microsoft Forms 2.0 MultiPage Display multiple pages Microsoft Forms 2.0 OptionButton Choose between multiple options Microsoft Forms 2.0 ScrollBar Basic horizontal and vertical scrollbars Microsoft Forms 2.0 SpinButton Button with Up and Down arrows Microsoft Forms 2.0 TabStrip Property-sheet style tabs for displaying multiple pages Microsoft Forms 2.0 TextBox Multiline text input and display Microsoft Forms 2.0 ToggleButton Button with two possible states Microsoft Web Browser Control View ActiveX documents such as Microsoft Word documents, Microsoft Excel spreadsheets, and so on

Select the Microsoft Forms 2.0 CommandButton Control from the list and select OK. The Control Pad brings up the Object Editor for letting you set the control's properties.

Using the Object Editor and Properties Window

Figure 11.2 shows the Object Editor invoked along with the Properties Window for editing the control and its properties.

Figure 11.2. Object Editor with the Microsoft Forms 2.0 CommandButton Control.

The control is shown on a grid, and the properties of the control are listed in the Properties window. You can adjust the size of the button by dragging the selection border drawn around the button. If you click on the button, you can edit the caption of the button. Many ActiveX controls support in-place editing. You can double-click on the button to toggle the display of the Properties window. You can also access the Properties window by right-clicking the control.

The Properties window lists the property names and values in a tabular form and enables you to edit them by double-clicking the property and clicking the Apply button. You can also set many properties by selecting appropriate values from the

drop-down list at the top.

After you finish setting the properties for the control, close the Object Editor window, and voil[ac]a! The ActiveX Control Pad inserts the appropriate HTML syntax for embedding the control using the <OBJECT> tag. The Control Pad locates the class-identifier for the control from the registry and automatically sets the CLASSID for you. The properties you have set for the control are specified in pairs of PARAM NAME and PARAM VALUE tags. Figure 11.3 shows the HTML file in the Text Editor with the <OBJECT> tag inserted by the Object Editor.

Figure 11.3. HTML file with embedded CommandButton ActiveX control.

The vertical pane to the left of the HTML file provides you with a visual cue of the objects inserted in the HTML file. You can see a cube icon in Figure 11.3 to the left of the <OBJECT> tag in the Text Editor. If you scroll through the file, the Control Pad automatically repositions the icon adjacent to the <OBJECT> tag. You can click the cube icon to launch the Object Editor again, whenever you want to revise the control's properties.

ActiveX Control Pad thus maps a visual interface on top of the stream metaphor of HTML.

Using the Script Wizard

To experience the ease of using the Script Wizard, first add a Text Box ActiveX Control to your HTML file. After adding the CommandButton control in the previous section, repeat the procedure just described to add a Text Box control to the HTML file. ActiveX Control Pad assigns default Object IDs of CommandButton1 and TextBox1 to the controls respectively. Set the CAPTION of CommandButton1 as Click Me. You can also customize the font face and size of the caption. After you have both the controls embedded in your HTML file, invoke the Script Wizard from the Tools menu or from the toolbar icon. The Script Wizard starts as shown in Figure 11.4.

Figure 11.4. The Script Wizard window.

The Script Wizard consists of three panes, with a command panel at the bottom.

The left pane is the *Event Pane*, which displays a tree view of the objects and events to which you can assign scripts. The objects are listed in alphabetical order by ID name, and the events of that object are listed under that branch of the tree. To assign an action to a particular event, locate the object and click the event that you want to script. Thus you can choose the event handler you want to script.

The right pane is the *Action Pane*, which displays the various actions you can perform and the properties you can change in response to each event. If you have global variables and procedures in the script on your page, they are also listed in the Action Pane. To associate an action with the event you have selected, double-click the action. If you want to change the property of some object, double-clicking brings up a dialog box in which you can enter the new property value.

To define new global variables or write procedures, right-click in the Action Pane and select the appropriate option from the pop-up menu.

The bottom pane is the *Script Pane*, which lists in a sequential manner the actions you have assigned to specific events, giving you a lucid interpretation of the actions you have selected. You can reorder the sequence of actions, insert and delete actions, and modify property values by using the command buttons in the Script Pane. The Script Pane displays this simple list of actions, called List View by default. However, if you are comfortable with writing and editing scripts, you can also display the event-handler script by selecting the Code View radio button.

You would use Code View when writing event-handlers that do the following:

- Invoke methods that have different numbers or names of arguments than the event handler itself.
- Require control of the flow of execution. Thus if you use conditional or looping statements, you have to use Code View.

Note

ActiveX Control Pad does not support scripting of HTML <AHREF="..."> or <FRAMESET> tags.

Select the MouseUp event of CommandButton1 in the Event Pane, and double-click the Text property of TextBox1 in the Action Pane. Enter Hello World in the dialog box that appears. Close the Script Wizard, and you can see the following VBScript code added to your HTML file:

```
<SCRIPT LANGUAGE="VBScript">

<!--

Sub CommandButton1_MouseUp(Button, Shift, X, Y)

TextBox1.Text = "Hello World!"

end sub

-->

</SCRIPT>
```

Save the file as HELLO.HTM and view it in Internet Explorer 3.0. When you press the button, the text box displays Hello World.

You can return to the Script Wizard for editing the event handler directly by clicking the yellow icon next to the <SCRIPT> tag. The Script Wizard displays the Event Pane with a solid icon next to the MouseUp event of CommandButton1 in the event hierarchy indicating the presence of an event handler. Script Wizard also adds the event handler to the Procedures in the Action Pane.

Switch to Code View to see the event-handler code in the Script Pane.

```
Sub CommandButton1_MouseUp(Button, Shift, X, Y)

TextBox1.Text = "Hello World!"
```

The Sub statement is shown in a title at the top with the body of the event handler underneath. Notice that no End Sub statement is displayed. Script Wizard adds the End Sub statement for you after you finish writing the body of the event handler.

Caution

When working in Code View, do not type the End Sub statement at the end of your event handler, as this will result in duplicate End Sub statements being added to the script.

You can change the display font of the Script Pane Code View by right-clicking in the Script Pane.

You have just used the ActiveX Control Pad to develop an interactive Web page, using ActiveX technology, without writing code, with point-and-click ease!

Script Wizard can generate both VBScript and JavaScript code. You can select the code it generates from the Tools|Options menu. However, you may not mix both the languages in the same HTML file. If you have scripts in both languages in your HTML file, when you start Script Wizard, it will assume the language of the first <SCRIPT> block to be the default scripting language of the entire HTML file.

Note

ActiveX Control Pad does not support scripts external to the HTML file using the SRC= attribute of the HTML <SCRIPT> tag.

HTML Layout Control

Traditional HTML lacks 2D layout specification. In other words, there is no way for the Web author to specify the exact placement of images, text, and objects on a HTML page. For the most part, the browser is in control, not the author. Frames and tables have enhanced the interface of Web sites, but they are a far cry from the advanced layout capabilities that have turned desktop publishing into an industry.

Authors need to be able to specify exact placement of text, images, and other controls on a Web page to create well-defined interfaces. For more sophisticated Web page design, they need to be able to overlap text and images, use transparency effects, and specify the layering of the objects that are placed on the Web page.

As you might expect, these capabilities are soon to be incorporated in the W3C HTML specification. The World Wide Web Consortium, which defines HTML standards, has published a preliminary draft specification for 2D layout capabilities by evolving the stylesheet and frameset specifications. Microsoft has given ActiveX Web authors a head start in this direction by developing a special ActiveX Control for 2D HTML layout.

HTML Layout Control: Overview

The HTML Layout Control is an ActiveX Control that acts as a container for other ActiveX controls. It specifies a 2D layout region within the HTML file in which it is placed. The HTML Layout Control uses a separate file for referencing the controls that are placed within the layout. This file is a simple text file with an .ALX extension and is stored along with the HTML file. The HTML Layout Control is inserted in the HTML file using the <OBJECT> tag.

Note

This is an early implementation of the 2D layout capabilities to be supported in future HTML specifications. These layout specifications will be incorporated into native HTML syntax and will not require a separate file. Microsoft has promised to adhere to the W3C standards and accordingly incorporate layout support natively in Microsoft Internet Explorer in future versions.

Multiple HTML layout regions can be incorporated into a single HTML file. Multiple instances of the HTML Layout Control are created, and they can be individually aligned or placed in tables, and so on. These Layout Controls behave independently of each other and do not support scripting across multiple layout regions. This means that events occurring in one layout region are not visible to controls in other layout regions within the same HTML document.

Using the 2D-Style Layout Editor

This section explores how to incorporate a 2D-style layout region in an HTML file using the ActiveX Control Pad.

Start the ActiveX Control Pad from the Windows Start menu and when the new HTML file is being displayed in the Text Editor, select Edit|Insert HTML Layout. In the dialog box that appears, you can select a name and location for the .ALX file that would contain the 2D layout information. This file would normally be located in the same directory as the source HTML file. Specify new.alx for the filename and Control Pad confirms whether you want to create a new file. Click OK, and the HTML layout control is inserted in the HTML file.

Click the icon next to the <OBJECT> tag, and the ActiveX Control Pad launches the HTML Layout Editor as shown in Figure 11.5.

Figure 11.5. The HTML Layout Editor.

The HTML Layout Editor consists of a 2D region in which you can place the controls shown on a floating toolbox. The toolbox contains the ActiveX controls that you can place within the layout. This is similar to designing a form in Visual Basic.

You can specify default properties like the background color of the Layout region by either right-clicking on the empty region or by selecting Properties from the View menu. You can place a control in the 2D region by selecting it from the Toolbox and drawing it on the region. The Properties window displays the properties of the control that has the focus. You can also access the properties of a particular control by right-clicking the control. Experiment with using the various controls provided with ActiveX Control Pad.

As you become comfortable with using the Layout Editor, you will soon realize the incredible power you have for creating an attractively designed Web page. You can design an interface with command buttons, labels, text boxes, and list boxes. You can determine the exact placement of images and text, overlap controls, and set the layering of controls. Some of the subtle properties that are very effective in creating good Web pages are

- BackStyle: used to add transparency effects.
- ControlTipText: specifies the tooltip text that appears when the user brings the mouse over the control.
- Picture: used to display images over controls such as buttons. Some controls also support tiling of the image.
- Accelerator: enables the user to navigate quickly using accelerator key combinations.
- MouseIcon: used to change the mouse pointer when the mouse is over the control.

After you finish drawing your controls, you can jump directly to the Script Wizard or you can return to Text Editor to view your HTML file. Control Pad will prompt you to save the file. The Text Editor window reappears, and the source HTML file now includes the <OBJECT> tag for the HTML Layout Control, as shown following:

```
<HTML>

<HEAD>

<TITLE>New Page</TITLE>

</HEAD>

<BODY>

<OBJECT CLASSID="CLSID:812AE312-8B8E-11CF-93C8-00AA00C08FDF"

ID="new_alx" STYLE="LEFT:0;TOP:0">

<PARAM NAME="ALXPATH" REF VALUE="new.alx">

 </OBJECT>

</BODY>
```

```
</HTML>
```

When the browser parsing this file encounters the <OBJECT> tag, it instantiates the HTML Layout Control. The HTML Layout Control loads the .ALX file specified in the ALXPATH parameter. The HTML Layout Control in conjunction with the browser then displays the controls embedded in the .ALX file using the exact 2D layout as specified by you, the author.

Note

The complete URL for the ALXPATH parameter should be specified unless the .ALX file resides in the same directory as the source HTML file or in a directory underneath it.

You can view the resulting page by opening the source HTML file in Internet Explorer 3.0.

Tip

You can change the alignment of the HTML Layout region within the source file by surrounding in regular HTML formatting tags (such as <CENTER> or ...) the <OBJECT> tag for the HTML Layout control. This is not achieved by setting the LEFT: and TOP: attributes of the STYLE parameter in the <OBJECT> tag used to insert the layout region.

Next, take a closer look at the .ALX file created for the Layout Control.

The .ALX File

The .ALX file is a text file and can be edited with a regular text editor. The .ALX file specifies the layout within the fixed 2D region—the controls embedded in it as well as the scripts associated with those controls. The layout is specified using the <DIV> tag, which is used by the W3C as a block tag for containing divisions in an HTML document. The <DIV> tag is used as shown in the following:

```
<DIV [ID=Layout_ID] STYLE = "layout-style-attributes">

   object-blocks

</DIV>
```

The <DIV> tag has two attributes.

- ID attribute, which is optional but is useful for referencing the layout control in scripts
- STYLE attribute, which specifies the style for the division

The STYLE attribute further has the following attributes:

- LAYOUT: This is defined as FIXED for the 2D layout region
- HEIGHT: The height of the layout region in pixels
- WIDTH: The width of the layout region in pixels
- BACKGROUND: The background color of the region in HEX digits

Inside the <DIV> tag are the <OBJECT> blocks specifying the controls placed within the 2D region.

Caution

The <OBJECT> tag may not be used to insert images, documents, or applets inside the .ALX file. The HTML Layout Control implementation currently supports only ActiveX Controls conforming to the ActiveX Controls '96 specification for windowless, transparent controls.

One or more <SCRIPT> blocks may be inserted before or after the <DIV> block, but not within it.

The limitation of this preliminary implementation of the layout region in the form of a separate .ALX file is that a transparent object hierarchy including the source HTML file and the controls within the layout region is no longer present.

The browser parses the HTML source file before the Layout Control renders the layout specified in the .ALX file. Hence, you cannot access any properties, events, or methods of the objects in the layout from scripts in the source HTML document.

Similarly, the scripts in the .ALX file cannot access all the methods and events of the window object of the HTML file. Scripts can only access the window.location.href property of the source HTML file so that the controls placed in the layout region can trigger navigation of the browser window.

Note

Some ways to bypass the inherent limitation of the .ALX layout implementation have been tried with mixed results, particularly for accessing the properties or methods of ActiveX Controls in the .ALX Layout file from the source HTML file.

In some circumstances, adding an OnLoad="InitALX" attribute to the <OBJECT> tag for the HTML Layout or referencing the ActiveX Control in the layout FILE.ALX as Control_ID.FILE.property may seem to work, but these are not officially supported and thus not recommended.

Using HTML Layout Control Effectively

The HTML Layout Control can be used to create a layout template that you can reuse in multiple Web pages. You might use this for making navigation toolbars, consistent interfaces for query forms, and style templates for electronic Web Magazines (Ezines).

You can achieve a dramatic improvement in the look of your Web pages by using transparent overlapping images. A basic understanding of images used in ActiveX Control Pad gives you the power to use these techniques effectively.

Several controls in the toolbox, such as the CheckBox, Command Button, and so forth, enable you to specify a PICTURE property. The PICTURE property specifies images that are *embedded* in the .ALX file. Embedded images of the .bmp, .cur, .wmf, .jpg, .gif, .ico image file types are read by the PICTURE property, converted into text format, and stored with the DATA attribute. You should consider the size of the images you select for the PICTURE property because using many large images will bloat the .ALX file. Use the PICTURE property for small icons and graphics so that the .ALX file is not large, and the user does not have to download a separate image file.

The other way you can use images on your Web page is using the Image Control. The Image Control is different than the other controls because it references the image URL specified in its PicturePath property at runtime. The PicturePath property of the Image Control can load .bmp, .jpg, .gif, .wmf format image files, but the user has to download the external image from the separate file at runtime.

Transparent images are ones in which a particular pixel color is rendered with the color of the background by the browser. Embedded images in ActiveX Control Pad are always transparent, whether the source file format supported transparency or

not. In the case of file formats that do not support transparency like .bmp and .wmf, the HTML Layout Control assumes the color of the lower left pixel of the image to be the transparent color. Transparency in external images is determined by the file format of the image. The GIF89a image specification supports transparent images, but .jpg does not.

Note

The HTML Layout Control does not support animated GIF89a files.

If your Web page uses controls that are large in dimensions, you might want to improve the performance of your Web page by increasing the size of the off-screen buffer used for painting your page. The number of pixels reserved for off-screen painting is specified by the DrawBuffer property of the HTML Layout Control. The default size allows for 32,000 pixels, which is fine for most Web pages. If you use larger images, however, you can specify a higher value.

You saw how the HTML Layout control in conjunction with the ActiveX Control Pad lets Web authors develop interactive Web pages with precise control placement. The next section consolidates your knowledge of the ActiveX Control Pad and the HTML Layout Control to develop a simple ActiveX puzzle.

ActiveX Web Puzzle

After seeing the power and ease of use of the ActiveX Control Pad, it is time to develop a sample application of this powerful tool. Remember the game in which the numbers 1 to 8 appear randomly in a 3*3 matrix, and the player has to sort them using the empty slot in the matrix? Figure 11.6 shows just such a puzzle developed with the ActiveX Control Pad supplied on the accompanying CD-ROM. You can copy the source files from the SOURCE\CHAP13\ directory on the CD-ROM.

Figure 11.6. A puzzle developed using ActiveX Control Pad.

Developing such an application for the Web is quite easy using the ActiveX Control Pad. Continue through the steps needed to develop the puzzle.

Implementation Strategy

Here's an overview of the technique required for implementing the puzzle. The game starts with initializing the digits in a random order on the game board. A mouse click on a button horizontally or vertically adjacent to the empty slot should move the button into the slot, and all other invalid mouse clicks should be ignored. At every click, the program needs to check whether the puzzle is solved. It will also provide for restarting the game whenever desired.

The straightforward way of using eight buttons and moving them at every mouse click leads to the complexity of tracking the coordinates of buttons. A trick to avoid this is to use nine buttons instead of eight and simply change their caption. An invisible button simulates the empty slot. On a valid mouse click, you change the Caption of the invisible button to that of the clicked button and toggle the visibility of the two buttons.

To track the progress of the puzzle, you will maintain an array of eight Boolean values. Each element of the array corresponds to a button, and its value depends on whether the button is in the proper place. After the initialization in a random state, you evaluate the state and initialize the array.

You also keep a count of the number of moves so that you can display the number of moves in which the puzzle was solved.

Creating the Layout

Start by creating the layout for the puzzle using the ActiveX Control Pad. In a new HTML file, insert an HTML Layout Control and name the file as PUZZLE.ALX. Insert nine CommandButton controls, arranging them in a matrix similar to that shown in Figure 11.6. Change the default IDs for the buttons to CB1, CB2, and so on up to CB9.

Add two labels for displaying the winning message and the number of clicks at the top and then change their default IDs to LblWon and LblClicks. Next draw the Restart command button below the button array, setting its ID to btnRestart.

For enhancing the aesthetics of this puzzle, you can add an Image Control for displaying a background image. Add an Image Control, setting its Picture property to point to an image file. If you have an image that is small in size, you can set the PictureTilingMode to True. The Image Control is drawn in front of the other controls, so right-click it and choose Send to Back from the pop-up menu. Now select the LblWon, btnRestart, and LblClicks controls and change their BackStyle property to transparent.

Tip

Press the Control key while clicking for selecting multiple non-adjacent controls so that you can edit properties of multiple controls at a time.

Adjust the BackColor and ForeColor of the buttons in the grid to a suitable color. Close the Layout Editor and save PUZZLE.ALX. Also save the source HTML file as PUZZLE.HTM.

The resulting source HTML file PUZZLE.HTM is shown in Listing 11.1.

Listing 11.1. The source HTML file for ActiveX Puzzle (PUZZLE.HTM).

```
<HTML>

<HEAD>

<TITLE>New Page</TITLE>

</HEAD>

<BODY>

<CENTER>

    <OBJECT ID="puzzle_alx"

     CLASSID="CLSID:812AE312-8B8E-11CF-93C8-00AA00C08FDF">

        <PARAM NAME="ALXPATH" REF VALUE="puzzle.alx">

    </OBJECT>

</CENTER>

</BODY>

</HTML>
```

You are done with the layout of your puzzle! Now you can add the scripting to make it functional.

Scripting the Puzzle

Before we embark on writing the script for the puzzle, make sure you are comfortable with using VBScript. If not, it might be a good idea to review Chapter 8, "VBScript," which discusses the VBScript language in detail.

While the PUZZLE.ALX is open in the Layout Editor, start Script Wizard by clicking on the Script Wizard Toolbar icon or by right-clicking in the background of the .ALX file and selecting "Script Wizard..." from the pop-up menu.

Start by defining the global variables to use. In the Action Pane right-click pop-up menu, choose Add Global Variable. Enter two variables in this way, named Click and Right(8). Click counts the number of moves, and Right is the array in which you maintain the progress of the game.

Next, script the custom procedures. Three procedures are used.

1. 1. Call_To_Start_Fresh() initializes the puzzle.
2. 2. Display_Buttons_At_Random() does just that.
3. 3. Is_Over() checks whether the puzzle is solved.

Choose Add Procedure from the Action Pane pop-up menu, and add the three procedures shown in Listing 11.2.

Listing 11.2. Procedures for the puzzle.

```
' This Subroutine is used to to Initialize the Game

Sub Call_To_Start_Fresh()

Dim t

' Array Right() is Initialized to False :0:

    for t=1 to 8

        Right = 0

    Next

' Subroutine Display_Buttons_At_Random is used to

' initially start the game in random state

Call Display_Buttons_At_Random()

' Make the Label for "You Have Won" Invisible

    LblWon.Visible = False

    LblClicks.Visible = False

    LblClicks.Caption = ""

    Click = 0

' To check whether the game has been won by randomize!
```

```
    Call Is_Over()
End Sub
' Subroutine Display_Buttons_At_Random is used to
' Initially Start the game in random state
Sub Display_Buttons_At_Random()
Dim Number(8)
Dim Random_Number
Dim Button
Dim j
Dim Re_Generate
Button=1
j=1
Re_Generate=True
while Button <= 8
    While Re_Generate = True
        Re_Generate = False
        Randomize
        Random_Number= Int((8 * Rnd)+1)
        while Re_Generate = False And j <= Button-1
            if Number = Random_Number then
                Re_Generate = True
            End if
            j = j + 1
        Wend
        j=1
        if Re_Generate = False then
            Number(Button) = Random_Number
```

```
            Button = Button + 1
        End If
    Wend
    Re_Generate = True
Wend
' Make all buttons visible accept the last
CB1.Visible = True
CB2.Visible = True
CB3.Visible = True
CB4.Visible = True
CB5.Visible = True
CB6.Visible = True
CB7.Visible = True
CB8.Visible = True
CB9.Visible = False
' Assign the generated random numbers to all buttons
CB1.Caption = CStr(Number(1))
CB2.Caption = CStr(Number(2))
CB3.Caption = CStr(Number(3))
CB4.Caption = CStr(Number(4))
CB5.Caption = CStr(Number(5))
CB6.Caption = CStr(Number(6))
CB7.Caption = CStr(Number(7))
CB8.Caption = CStr(Number(8))
end sub
' Subroutine to check whether the puzzle is solved
Sub Is_Over()
```

```
Dim Total

Dim Num

If CB1.Caption = "1" then

    Right(1) = 1

End If

If CB2.Caption = "2" then

    Right(2) = 1

End If

If CB3.Caption = "3" then

    Right(3) = 1

End If

If CB4.Caption = "4" then

    Right(4) = 1

End If

If CB5.Caption = "5" then

    Right(5) = 1

End If

If CB6.Caption = "6" then

    Right(6) = 1

End If

If CB7.Caption = "7" then

    Right(7) = 1

End If

If CB8.Caption = "8" then

    Right(8) = 1

End If

for i=1 to 8
```

```
    Total = Total + Right
Next
If Total = 8 then
    LblWon.Visible = True
    btnRestart.Caption = "Play Again  ?"
    If Click <> 0 then
        LblClicks.Visible = True
        LblClicks.Caption = "          in " & CStr(Click) & " clicks"
    End If
End If
End Sub
```

After entering the procedures, you'll add the event handlers for the MouseUp event of the buttons in the grid. Here's what the event handler does:

- Checks whether the button is in place and updates the score.
- Checks whether an adjacent button is invisible and toggles the visibility of both buttons, while transferring its Caption.

The number of possible moves for a button is dependent on its location in the grid, for example, CB1 can move only to CB2 and CB4, whereas CB2 can move to CB1, CB5, and CB3. The third possibility is the center button, for which there are four possible moves.

The event handler for the button CB5 is shown in Listing 11.3; it is pretty simple to add similar ones for the others.

Listing 11.3. Event handlers for MouseUp event of buttons in puzzle.

```
Sub CB5_MouseUp(Button, ShIft, X, Y)
If CB5.Caption = "5" then
    Right(5) = 1
End If
If CB5.Visible = True then
    If CB2.Visible = False then
        CB2.Visible = True
        CB2.Caption = CB5.Caption
        CB5.Caption = ""
```

```
        CB5.Visible = False
        Click = Click + 1
    End If
    If CB4.Visible = False then
        CB4.Visible = True
        CB4.Caption = CB5.Caption
        CB5.Caption = ""
        CB5.Visible = False
        Click = Click + 1
    End If
    If CB6.Visible = False then
        CB6.Visible = True
        CB6.Caption = CB5.Caption
        CB5.Caption = ""
        CB5.Visible = False
        Click = Click + 1
    End If
    If CB8.Visible = False then
        CB8.Visible = True
        CB8.Caption = CB5.Caption
        CB5.Caption = ""
        CB5.Visible = False
        Click = Click + 1
    End If
End If
If CB5.Caption = "5" then
    Right(5) = 1
```

```
else

    Right(5) = 0

End If

Call Is_Over()

End Sub
```

Finally, you call the initialization procedure from the OnLoad event of the HTML Layout Control and also handle the MouseUp event for the Restart button as shown in Listing 11.4.

Listing 11.4. HTML Layout OnLoad and Restart button MouseUp event handlers.

```
' Subroutine called when the Layout is loaded

Sub Layout1_OnLoad()

Call Call_To_Start_Fresh()

End Sub

' Subroutine to be called when Button for "Re-start" or "Play Again" is Pressed

Sub btnRestart_MouseUp(Button, Shift, X, Y)

Call Call_To_Start_Fresh()

LblWon.Visible = False

btnRestart.Caption = "Restart"

End Sub
```

After you have the scripts in place, save the files PUZZLE.ALX and PUZZLE.HTM, and double-click PUZZLE.HTM to start Microsoft Internet Explorer. Figure 11.7 shows the winning screen after the puzzle is solved.

Figure 11.7. The ActiveX Puzzle winning screen.

Enjoy the game!

ActiveX Search Page

The ActiveX Puzzle application demonstrates the power of the Layout Control in creating interactive Web pages. Now see how you can use the ActiveX Control Pad to utilize the Internet Explorer Object Model interface for adding advanced capabilities to your Web pages.

A fairly common Web application is an all-in-one Internet search page where the user can type his keywords and select the source and/or type of search. Normal HTML forms that accomplish this generally present a cluttered interface to the user, because they are limited by the HTML form object implementation.

Using ActiveX, however, you can develop a sophisticated interface using ActiveX controls in HTML layout regions, and

create an intuitive interface for the user. Your search page needs to collect data from the user and pass it on to the search engine of the user's choice.

To demonstrate how you can accomplish this using the ActiveX Control Pad, you'll develop a sample Search Page from which the user can select the search engine from a drop-down list box and enter his query keywords. When he activates the Go button, your Web page will pass the keywords along with any optional parameters to the appropriate search engine server. The browser will navigate to the site and display the query results on that site. The ActiveX Search Page as seen in Internet Explorer is shown in Figure 11.8. The source files for the ActiveX Search Page can be found on the CD-ROM accompanying this book in the \SOURCE\CHAP13 directory.

Figure 11.8. The ActiveX Search Page.

For your sample search page, allow the user to choose his search site between Yahoo, WebCrawler, DejaNews, and AltaVista. Note that your Web page has to supply the query data to the server script executing on these Web sites. In order to interface your page correctly with the server application on the search site, emulate the form used by that search site. Further, some search sites support additional parameters for customizing the results. For instance, the popular DejaNews Usenet search enables you to specify the number of hits requested, and AltaVista enables you to return results in Standard, Compact, or Detailed form.

Note

The server applications of search engines are modified from time to time, and if you are interested in implementing an ActiveX search interface on the Web, you should constantly monitor the sites accordingly.

Emulating an HTML Form Through ActiveX

The HTML FORM object lies below the Document Object in the Object Scripting Model. The HTML form that you create in your source file would have to be filled with the data from the controls in the HTML Layout Control. You create hidden input fields in your HTML form and transfer the data using scripts. When the user clicks the Go Button, you trigger the SUBMIT event of the form object.

Because each search engine uses a different FORM, you include an HTML form for each of them, with the appropriate number of input fields, which are given specific names used by the search engine. The trick to discover the number and names of input fields required by a search engine is to view the HTML source of the query page of the Web site.

To demonstrate the advantages of using ActiveX to create a form rather than using pure HTML forms, the next step is to alter the interface at runtime according to the search engine selected by the user. If this is done, the user easily can utilize the custom options that a particular search engine supports.

Developing the ActiveX Search Page

The source for the HTML file with unique forms for each search engine is shown in Listing 11.5.

Listing 11.5. Source HTML file for ActiveX Search page (SEARCH.HTM).

```
<HTML>
```

```
<HEAD>
<TITLE>ActiveX Search Page</TITLE>
</HEAD>
<BODY>
<!-- form for search in DejaNews -->
<FORM NAME="SearchFormIdDeja" METHOD=POST
ACTION="http://xp5.dejanews.com/dnquery.xp">
    <INPUT TYPE="hidden" NAME="query">
    <INPUT TYPE="hidden" NAME="defaultOp" VALUE="AND">
    <INPUT TYPE="hidden" NAME="svcclass" VALUE="dncurrent">
    <INPUT TYPE="hidden" NAME="maxhits">
</FORM>
<!-- form for search in AltaVista -->
<FORM NAME="SearchFormIdAlta" METHOD=GET
ACTION="http://www.altavista.digital.com/cgi-bin/query">
    <INPUT TYPE="hidden" NAME="pg" VALUE=q>
    <INPUT TYPE="hidden" NAME="what">
    <INPUT TYPE="hidden" NAME="fmt">
   <INPUT TYPE="hidden" NAME="q">
</FORM>
<!-- form for search in Yahoo -->
<FORM NAME="SearchFormIdYahoo" METHOD=GET
ACTION="http://search.yahoo.com/bin/search">
    <INPUT TYPE="HIDDEN" NAME="p">
</FORM>
<!-- form for search in WebCrawler -->
<FORM NAME="SearchFormIdCrawler"  METHOD="POST"
```

```
ACTION="http://www.webcrawler.com/cgi-bin/WebQuery" >

    <INPUT TYPE="hidden" NAME="searchText">

    <INPUT TYPE="hidden" NAME="maxHits">

    <INPUT TYPE="hidden" NAME="mode">

</FORM>

<!-- HTML Layout for ActiveX Search Page -->

<OBJECT CLASSID="CLSID:812AE312-8B8E-11CF-93C8-00AA00C08FDF"

ID="query_alx" STYLE="LEFT:0;TOP:0">

<PARAM NAME="ALXPATH" REF VALUE="query.alx">

</OBJECT>

</BODY>

</HTML>
```

Creating the Layout for the ActiveX Search Page

The complete layout is illustrated in Figure 11.9.

Figure 11.9. Layout for the ActiveX Search page.

The layout development for the ActiveX Search page is fairly straightforward. Table 11.2 enlists and describes the controls used in the layout.

Table 11.2. Controls used in the ActiveX Search page.

Control_ID Description btnGo Command Button for starting the search lblEngine "Select Engine" Label lblGet "Get" Label lblHits "Hits" Label lblQuery "Enter Query" Label lblReturn "Return:" Label lblSearch "Search:" Label lblTitle "ActiveX Search Page" Label lbNames Drop-down list of Web Search Engines optbtnHitType1 Option (Radio) Button for selecting Format of Hits optbtnHitType2 Option (Radio) Button for selecting Format of Hits optbtnHitType3 Option (Radio) Button for selecting Format of Hits optbtnNews Option (Radio) Button for selecting Usenet search optbtnWeb Option (Radio) Button for selecting Web search txtboxHits Text Box for specifying number of hits to return txtboxQuery Text Box for entering the query text

A colorful background is selected for lblTitle, and the labels are made transparent for an enhanced appearance. You are now ready for adding scripts.

Scripting the ActiveX Search Page

You'll proceed by first adding the global variables as before. Add query_string, mode_str, maxhits_str, what_str = "Web" as

four global variables, from the pop-up menu in the Action Pane. The global variables are used as follows:

- query_string contains the text of the query.
- mode_str defines the type of hits desired for WebCrawler and AltaVista searches.
- maxhits_str specifies the maximum number of hits to return for WebCrawler and DejaNews searches.
- what_str selects Web or Usenet search for AltaVista.

The HTML Layout OnLoad event handler adds the names of the search engines to the list box at runtime and selects the first as default. This is shown following:

```
Sub Layout1_OnLoad()
    call lbNames.AddItem("Yahoo")
    call lbNames.AddItem("Web Crawler")
    call lbNames.AddItem("DejaNews UseNet")
    call lbNames.AddItem("Altavista")
    lbNames.Value = "Yahoo"
end sub
```

The script for handling the Change event for the drop-down list box, radio buttons, and text boxes is shown in Listing 11.6.

Listing 11.6. Change event handlers for ActiveX Search page controls.

```
Sub lbNames_Change()
 optbtnHitType1.Visible = True
 Dim comp
 comp = StrComp(lbnames.Value,"WebCrawler")
 if comp = 0 then
     optbtnHitType1.Caption = "Compact"
     optbtnHitType1.Visible = True
     optbtnHitType2.Caption = "Titles"
     optbtnHitType2.Visible = True
     optbtnHitType3.Caption = "Summaries"
     optbtnHitType3.Visible = True
     optbtnHitType1.Value = True
     optbtnWeb.Visible = False
```

```
    optbtnNews.Visible = False
    optbtnHitType3.Value = True
    lblReturn.Visible = True
    lblGet.Visible = True
    lblHits.Visible = True
    txtboxHits.Visible = True
    lblSearch.Visible = False
End If
comp = StrComp(lbnames.Value,"Yahoo")
if comp = 0 then
    optbtnHitType1.Visible = False
    optbtnHitType2.Visible = False
    optbtnHitType3.Visible = False
    optbtnWeb.Visible = False
    optbtnNews.Visible = False
    lblReturn.Visible = False
    lblGet.Visible = False
    lblHits.Visible = False
    txtboxHits.Visible = False
    lblSearch.Visible = False
End If
comp = StrComp(lbnames.Value,"DejaNews UseNet")
if comp = 0 then
    optbtnHitType1.Visible = False
    optbtnHitType2.Visible = False
    optbtnHitType3.Visible = False
    optbtnWeb.Visible = False
```

```
    optbtnNews.Visible = False
    lblReturn.Visible = False
    lblGet.Visible = True
    lblHits.Visible = True
    txtboxHits.Visible = True
    lblSearch.Visible = False
 End If
 comp = StrComp(lbnames.Value,"Altavista")
 if comp = 0 then
    optbtnHitType1.Caption = "Standard"
    optbtnHitType1.Visible = True
    optbtnHitType2.Caption = "Compact"
    optbtnHitType2.Visible = True
    optbtnHitType3.Caption = "Detailed"
    optbtnHitType3.Visible = True
    optbtnHitType1.Value = True
    optbtnWeb.Visible = True
    optbtnNews.Visible = True
    lblReturn.Visible = True
    lblGet.Visible = False
    lblHits.Visible = False
    lblSearch.Visible = True
    optbtnWeb.Value = True
    txtboxHits.Visible = False
 End If
end sub
Sub optbtnHitType3_Change()
```

```
 Dim comp
 comp = StrComp(lbnames.Value,"Altavista")
 if comp = 0 then
     mode_str = "d"
 else
     mode_str = optbtnHitType3.Caption
 End If
end sub
Sub optbtnHitType2_Change()
 Dim comp
 comp = StrComp(lbnames.Value,"Altavista")
 if comp = 0 then
     mode_str = "c"
 else
     mode_str = optbtnHitType2.Caption
 End If
end sub
Sub optbtnHitType1_Change()
 Dim comp
 comp = StrComp(lbnames.Value,"Altavista")
 if comp = 0 then
     mode_str = "."
 else
     mode_str = optbtnHitType1.Caption
 End If
end sub
Sub txtboxQuery_Change()
```

```
    query_string = txtboxQuery.Value

end sub

Sub txtboxHits_Change()

    maxhits_str = txtboxHits.Value

end sub
```

The Change event handlers for the type of results desired adapt the mode_str variable to the appropriate value. The Change event handlers for the txtboxQuery and txtboxHits Text boxes update the query_string and maxhits_str global variables respectively.

The search engine drop-down list box determines the controls that are visible after a particular search engine is selected. The Yahoo search doesn't support any parameters; WebCrawler enables you to select the format of the results and the number of hits. Similarly, DejaNews supports specifying the number of hits returned, whereas with AltaVista you choose the format of the results and the search field (Web or Usenet).

The lbNames_Change event handler makes the appropriate controls visible when a particular search engine is selected. Selection among the type of results desired (mode_str) is implemented by handling the Change event for each of the option buttons, whereas the field of query for the AltaVista search is scripted using MouseUp event-handlers. This is because mode_str is used by both WebCrawler and AltaVista, whereas the Web/Usenet choice of option buttons is only relevant to AltaVista.

Finally, the MouseUp event handlers for the AltaVista Usenet/Web Option Buttons and the Go Button are given in Listing 11.7.

Listing 11.7. MouseUp event handlers for ActiveX Search page controls.

```
Sub optbtnNews_MouseUp(Button, Shift, X, Y)

    what_str = "news"

end sub

Sub optbtnWeb_MouseUp(Button, Shift, X, Y)

    what_str = "web"

end sub

Sub btnGo_MouseUp(Button, Shift, X, Y)

 Dim frmSearchFormIdDeja

 Dim frmSearchFormIdYahoo

 Dim frmSearchFormIdCrawler

 Dim frmSearchFormIdAlta

 Dim comp
```

```
comp = StrComp(lbnames.Value,"DejaNews UseNet")
if comp = 0 then
    Set frmSearchFormIdDeja = Document.SearchFormIdDeja
    frmSearchFormIdDeja.query.Value = query_string
    frmSearchFormIdDeja.maxhits.Value = maxhits_str
    frmSearchFormIdDeja.Submit
End If
comp = StrComp(lbnames.Value,"Yahoo")
if comp = 0 then
    Set frmSearchFormIdYahoo = Document.SearchFormIdYahoo
    frmSearchFormIdYahoo.p.Value = query_string
    frmSearchFormIdYahoo.Submit
End If
comp = StrComp(lbnames.Value,"Web Crawler")
if comp = 0 then
    Set frmSearchFormIdCrawler = Document.SearchFormIdCrawler
    frmSearchFormIdCrawler.searchText.Value = query_string
    frmSearchFormIdCrawler.mode.Value = mode_str
    frmSearchFormIdCrawler.maxHits.Value = maxhits_str
    frmSearchFormIdCrawler.Submit
End If
comp = StrComp(lbnames.Value,"AltaVista")
if comp = 0 then
    Set frmSearchFormIdAlta = Document.SearchFormIdAlta
    frmSearchFormIdAlta.what.Value = what_str
    frmSearchFormIdAlta.fmt.Value = mode_str
    frmSearchFormIdAlta.q.Value = query_string
```

```
    frmSearchFormIdAlta.Submit

 End If

end sub
```

The MouseUp event handler for the Go button checks for the string in the search engine drop-down list box, and accordingly it sets the hidden form variables to the global variables. After the required fields of the form are initialized, it calls the SUBMIT method for the form, submitting the query to the search engine.

You can test the ActiveX Search Page by copying the source files to a Web server and browsing the SEARCH.HTM file using Internet Explorer 3.0.

This example shows you the ready-made functionality that the ActiveX Controls supplied with the ActiveX Control Pad offer for developing customized superior interfaces for your Web pages. Now take a cursory look at some additional controls that Microsoft provides for developing superlative Web pages.

Additional ActiveX Controls Supplied by Microsoft

The ActiveX Controls shipped with the ActiveX Control Pad are standard controls that can be used for most application interfaces ported to the Web. However, to develop powerful compelling Web pages, you can make use of additional ActiveX Controls that are offered by Microsoft. These controls can be downloaded from the Internet at Microsoft's ActiveX Controls Web Page at http://www.microsoft.com/intdev/controls/ctrlref.htm. Table 11.3 lists sample controls available on the ActiveX Controls Web Page.

Table 11.3. Additional ActiveX Controls available at Component Gallery.

Control Description ActiveMovie Displays sound and video data with support for different file formats and streaming Animated Button Used for displaying various frame sequences of an AVI depending on the button state Chart Displays charts in different styles like bar-chart, pie-chart, and so on Gradient Draws shaded transition between colors Label Draws text in angles and along user-defined curves Marquee Displays scrolling and/or bouncing URLs Menu Displays a menu button or a pull-down menu Pop-up Menu Displays a pop-up menu Pop-up Window Displays specified HTML documents in a pop-up window Preloader Downloads a URL in the background to store in cache Stock Ticker Displays changing data in text or XRT format at regular intervals Timer Generates periodic events View Tracker Detects whether control is in the viewable area

Along with the above ActiveX controls, there are several controls from third-party vendors which are available at the ActiveX Component Gallery on the World Wide Web. The ActiveX Component Gallery at http://www.microsoft.com/activex/controls is an impressive assortment of powerful controls.

The next section addresses a few issues relevant to publishing Web pages developed using ActiveX Controls supplied along with ActiveX Control Pad or from third-party vendors.

Using Custom or Third-Party ActiveX Controls

There are over a thousand ActiveX Controls available today, and you can expect this figure to rise astronomically as ActiveX gains momentum. In order to use these custom controls to publish active Web pages, there is an important property that has to be specified in the <PARAM> attribute of the <OBJECT> tag used for the control. This is the CODEBASE property.

When users browse a Web page developed with an ActiveX Control that is not installed on their system, the CODEBASE

property identifies the source URL where the browser can locate the ActiveX Control and automatically download it. Depending on the browser's security settings, the browser will either ask the user before installing the control or install the control automatically and then display the Web page.

If you are using one of the controls shipped along with the ActiveX Control Pad, you can download the Mspert10.cab file from Microsoft's Web site at http://activex.microsoft.com/controls/mspert10.cab. Copy this file to the Web server on which you are publishing your Web page. To set the browser for automatic downloading, you need to point the CODEBASE attribute of the controls that you use to point to this file on your server. Alternatively, you can set CODEBASE to point to Microsoft's Web site, but this would require additional overhead for the browser to establish a connection with Microsoft's Web server.

Caution

If you use ActiveX controls on your server and point the CODEBASE attribute to your server, be sure that the control is digitally signed and certified. Refer to the Microsoft ActiveX Code Signing Documentation in the ActiveX Software Development Kit (SDK) for details.

When using ActiveX controls from the Component Gallery or a third-party ActiveX Control, be sure to refer to the documentation of the control and specify the CODEBASE attribute correctly. Otherwise, users who do not have the control installed on their system may be unable to view your Web page.

Some examples of ActiveX Controls shipping today are

- Macromedia's ActiveShockwave plays multimedia movies developed in Macromedia authoring tools like Director.
- Adobe Acrobat displays documents in Adobe Acrobat format.
- Black Diamond Consulting's SurroundVideo displays 360 degrees interactive panoramic view.
- Progressive Networks Real Audio Player plays streaming audio over the Internet at low-bandwidth connections.

You can download and look for many more ActiveX Controls in the ActiveX Component Gallery at http://www.microsoft.com/activex/controls. New controls are being added to the ActiveX Component Gallery frequently.

These examples should give you a glimpse of the exciting possibilities offered by ActiveX technology.

Summary

This chapter gave you a glimpse of how you can exploit the power of ActiveX using the first development environment for using ActiveX Controls to build Web pages, the ActiveX Control Pad.

In the next chapter, you'll learn how to build advanced Web pages, including animation, using the techniques that were discussed in Part 2, "ActiveX Scripting," and 3, "Active Web Pages," of this book.

Chapter 12

Advanced Web Page Creation

Creating Web pages with basic HTML is a fairly straightforward task, but adding active content with multiple frames is a different matter altogether. The secret is just to take things one step at a time, building content in stages. In this chapter, you'll learn how easy it can be to create exciting, complex Web pages.

Creating a Basic Web Site

This chapter presents a simple, frames-based Web site. You'll have a home page, composed of frames, that displays various other HTML pages within the frames. It contains some useful navigation elements to let users jump from one URL to another within a consistent browsing environment. To accomplish this, you'll develop a home page that uses two frames that remain static, along with a third that changes URLs to display information requested by the user. Figure 12.1 shows the basic concept for the frames.

Figure 12.1. The conceptualization of the frames layout for the sample home page.

During the conceptualization phase, I decided that the top frame, or banner frame, would be completely static and non-resizable, and that this frame would contain bitmaps used for navigation among topics. I also decided that the left frame, or contents pane, would be resizable and would display a more in-depth selection of topics in a textual format. This design leaves the frame to the right of the resizable frame border for use as a topic viewing area, where the various HTML files can be displayed.

Multiple HTML Files and Frames

You will learn how to integrate several HTML files into a master frames page that implements the concept shown in Figure 12.1. The frames page acts as a container for other HTML files and as the home page for the small sample Web site that you'll build throughout the course of this chapter. The files you'll start out with are:

- A banner page to display a logo and a navigation imagemap (BANNER.HTM).
- A contents page to provide textual navigation to other pages (TOC.HTM).
- A welcome page that welcomes users to the site (WELCOME.HTM).
- The main frames page (INDEXFRM.HTM), which is the main component of the site.

Stage 1: Creating the Basic Frames Page (INDEXFRM.HTM)

As you saw, the frames page for your sample Web site is to be made up of three frames. These frames are defined by two *framesets*, one horizontal and one vertical. A frameset is simply a container for frames that allows you to specify their size and orientation. Each frame in a frameset has a Universal Resource Locator (URL). This means that you can click a link in one frame and view the corresponding data in another frame. Each frame in the sample Web site you build in this chapter displays a separate HTML file. The banner and contents frames remain constant, but the topic frame displays various HTML documents.

A frame can be scrolling or nonscrolling (scrolling is the default), and resizable or static (resizable is the default). The following is the syntax for a basic frame:

```
<frame src=[URL] name=[FrameName] scrolling=[yes/no] [noresize]>
```

As mentioned earlier, two framesets are used for this page, one nested inside the other. The first frameset uses the rows attribute, which means it's a horizontal frameset composed of rows of frames. For the frame sizing in this preliminary stage, the frame in the top row is set to 10% of the browser window's height, and the bottom frame is set to the remaining 90% of the browser window, like this:

```
<frameset rows="10%,90%">
```

The second frameset uses the cols attribute, which means it's a vertical frameset composed of columns of frames. The initial sizing for the frame in the left column is set to 25% of the browser window's width, and the bottom frame is set to the remaining 75% of the browser window, like this:

```
<frameset cols="25%,75%">
```

Note

As new pages are developed for display within the various frames, the spacing used for the framesets will change (as the needs of the page evolve). The initial frameset values of rows="10%,90%" and cols="25%,75%" are just estimates at this stage.

The file INDEXFRM.HTM contains all of the code for the basic frameset used in the same Web site. The code for this is straightforward, as you can see in Listing 12.1.

Listing 12.1. The initial HTML source code for a basic nested frameset page with three frames

(INDEXFRM.HTM).

```
<html>
<head>
<title>Sample Web Site With Multiple Frames</title>
</head>
<!-- Set up the frames for the home page -->
<frameset rows="10%,90%">
   <frame src="banner.htm" name="banner" scrolling="no" noresize>
   <frameset cols="25%,75%">
      <frame src="toc.htm" name="contents">
      <frame src="welcome.htm" name="topic">
   </frameset>
   <!-- If the browser doesn't support frames, display this -->
   <noframes>
      <BODY>
      <p>
      This site uses frames, but your browser doesn't support them.
      </p>
      </body>
   </noframes>
</frameset>
</html>
```

In this case, the frame named "banner" is listed right after the first frameset statement, which means it inhabits the 10% spot (the top). The 90% spot (the bottom) is taken by the nested frameset, with the frame

named "contents" taking the 25% spot (the left), and the frame named "topic" taking the 75% spot (the right). Make sense? Take a look at the resulting sample home page shown in Figure 12.2, in its first incarnation.

Note

The files used to generate the version of the sample home page seen in Figure 12.2 can be found on the companion CD-ROM in the SOURCE\CHAP14\STAGE1 directory.

Figure 12.2. The first stage of the sample home page.

The Basic Frame URLs

Placeholder HTML files must be created in order for the frames to display as in Figure 12.2. The source files for the URLs used within the frames are quite simple at this point, although you'll develop each into a more complex version as you go along. The source code for each of the basic frame URLs is listed in the next sections.

The Banner Frame

The banner frame will eventually contain a sample logo and a series of images that act as links for graphical navigation. The source for the basic banner page, BANNER.HTM, is shown in Listing 12.2.

Listing 12.2. The initial HTML source code for the basic banner frame (BANNER.HTM).

```
<HTML>

<HEAD>

<TITLE>Banner</TITLE>

</HEAD>

<BODY>

<H2>Banner Frame</H2>

</BODY>

</HTML>
```

The Contents Frame

The contents frame will contain a bitmap and some text links to the topics to display in the topic frame. The source for the basic contents page, TOC.HTM, is shown in Listing 12.3.

Listing 12.3. The initial HTML source code for the basic contents frame (TOC.HTM).

```
<HTML>

<HEAD>

<TITLE>Contents</TITLE>

</HEAD>

<BODY>

<H2>Contents Frame</H2>

</BODY>

</HTML>
```

The Topic Frame

The topic frame will display various different HTML files, depending on what the user chooses in the contents frame or in the banner frame's navigation bar. The source for the placeholder topic file WELCOME.HTM is shown in Listing 12.4. Later you'll add two more pages, TOPIC1.HTM, and TOPIC2.HTM. These three pages will always appear in the topic frame.

Listing 12.4. The initial HTML source code for the basic welcome page (WELCOME.HTM).

```
<HTML>

<HEAD>

<TITLE>Welcome</TITLE>

</HEAD>
```

```
<BODY>

<H2>Welcome Page</H2>

</BODY>

</HTML>
```

As you can see, all of these pages are very similar at this point, being little more than skeletal HTML files generated by the ActiveX Control Pad (see Chapter 11, "Using ActiveX Control Pad," for more information). Next, you'll develop each page to show how the entire home page comes together with frames.

Stage 2: Developing the Banner Page (BANNER.HTM)

Now you can turn your attention to developing the banner. A sample logo image is needed, along with other images for navigating to the three topic URLs. Images for your pages can be created with a variety of tools. For these examples, I'll use the Persistence of Vision ray tracer (POVRAY) to create some beautiful images, along with Corel's Photo-Paint 6 for image manipulation and conversion.

Note

Be sure to visit the POVRAY Web site at http:\\www.povray.org to truly appreciate the incredibly beautiful and realistic images that can be generated with this *completely free* software! The new POVRAY 3.0 has been released for many platforms including: MS-DOS, Windows 3.1x with Win32s, Windows 95, Windows NT, Linux, UNIX, Macintosh, and SunOS.

The idea for the banner is to have a black background with a sample logo on the left, and three shiny stone spheres next to the logo, one after another, on the right. The stones are the navigation images corresponding to the Welcome page, the Topic 1 page, and the Topic 2 page.

The Banner Images

The Sample logo was created completely in Photo-Paint 6, using a plasma cloud texture fill for the background. Because the background for the banner frame is to be black, I blended the plasma fill gradually from solid black on the right using the Photo-Paint airbrush tool. Next, I added the word "Sample" (pretty creative, eh?) with the text tool. Finally, for a bit of drama, I added a slight lens flare with the lens flare tool. I centered the lens flare on the "p" in "Sample " and saved the image as SAMPLE.GIF.

The other banner images were created using POVRAY to generate the wonderful stone spheres, followed by Photo-Paint to add the text for each. The images were then saved as STONE1.GIF, STONE2.GIF, and

STONE3.GIF, and all four banner images are shown, in Photo-Paint, in Figure 12.3.

Figure 12.3. The images used in the Banner page, manipulated in Corel Photo-Paint 6.

Positioning the Images

Now that you have your images, continue by positioning them in the page. You want to have them aligned at the left and top, and the page background should be black. To accomplish this, the following code is used in the <BODY> tag for the page:

```
<BODY BGCOLOR="#0" TOPMARGIN="0" LEFTMARGIN="0">
```

The background color #0 is, of course, black, and the TOPMARGIN and LEFTMARGIN attributes are set to zero so the images will snuggle up close to the left and top of the frame in Internet Explorer (Netscape Navigator is a bit different, as you'll see). Next, the images are listed one after another, with no line or paragraph break codes, like this:

```
<img src="../gif/sample.gif" width=142 height=46>
<img src="../gif/stone1.gif" width=100 height=46 BORDER=0>
<img src="../gif/stone2.gif" width=100 height=46 BORDER=0>
<img src="../gif/stone3.gif" width=100 height=46 BORDER=0>
```

```
<A HREF="topic1.htm" TARGET="topic" ALT="Topic 1">
```

This line references TOPIC1.HTM as the link, and the Topic frame as the target. It's that simple!

Note

The target frame specified in the TARGET attribute is case-sensitive! The link to the target frame won't work if the case of the characters isn't exactly the same as specified in the page that defines the frame names (in this case, INDEXFRM.HTM).

There is already a basic Welcome page (WELCOME.HTM), but the Topic 1 and Topic 2 pages don't yet exist. By simply copying the Welcome page twice, naming the files TOPIC1.HTM and TOPIC2.HTM, and

changing the title and heading to Topic 1 and Topic 2, respectively, you have all the pages the sample site will use.

Listing 12.5 shows the final banner page source code. In this final incarnation, the code supports links to TOPIC1.HTM, TOPIC1.HTM, and TOPIC1.HTM from the sphere images (STONE1.GIF, STONE2.GIF, AND STONE3.GIF, respectively).

Listing 12.5. The final HTML source code for the Banner page (BANNER.HTM).

```
<html>

<head>

<title>Banner</title>

</head>

<BODY BGCOLOR="#0" TOPMARGIN="0" LEFTMARGIN="0">

<img src="../gif/sample.gif" width=142 height=46>

<A HREF="welcome.htm" TARGET="topic" ALT="Welcome">

<img src="../gif/stone1.gif" width=100 height=46 BORDER=0></a>

<A HREF="topic1.htm" TARGET="topic" ALT="Topic 1">

<img src="../gif/stone2.gif" width=100 height=46 BORDER=0></a>

<A HREF="topic2.htm" TARGET="topic" ALT="Topic 2">

<img src="../gif/stone3.gif" width=100 height=46 BORDER=0></a>

</body>

</html>
```

Adjusting the Frame Size

Now that the Banner page is complete, you can move on, right? Not so fast! The banner frame marginwidth and marginheight attributes must be set to zero in INDEXFRM.HTM, so the images snuggle up against the left and top of a frame when viewed in Netscape Navigator, like this:

```
<frame src="banner.htm" name="banner" scrolling="no" noresize
   marginwidth="0" marginheight="0">
```

I originally set up the frames to use percentages in the INDEXFRM.HTM file, and as the browser window resizes, so do the frames. Because the SAMPLE.GIF image uses bright colors along its bottom edge, an unacceptably sharp delineation occurs between the black background and the image's bottom edge when the browser window is maximized. To prevent this, you need to set the banner frame to a specific height: that of the images themselves.

The images are all 46 pixels high, as you can see from the height attributes in the code. After viewing the INDEXFRM.HTM file in both Netscape Navigator and Internet Explorer, I decided on a height of 48 pixels, taking into account the width of the frame borders, which in Internet Explorer can be adjusted, like this:

```
<frameset rows="48,*" frameborder="1" framespacing="1">
```

The statement rows="48,*" tells the browser to make the top frame of the frameset exactly 48 pixels high, and to use the rest of the window for the bottom frame. These changes give you the final version of the INDEXFRM.HTM file, shown in Listing 12.6.

Listing 12.6. The final HTML source code for the frames home page (INDEXFRM.HTM).

```
<html>
<head>
<title>Sample Web Site With Multiple Frames</title>
</head>
<!-- Set up the frames for the home page -->
<frameset rows="48,*" frameborder="1" framespacing="1">
   <frame src="banner.htm" name="banner" scrolling="no" noresize
      marginwidth="0" marginheight="0">
   <frameset cols="33%,69%">
      <frame src="toc.htm" name="contents">
      <frame src="welcome.htm" name="topic">
   </frameset>
```

```
   <!-- If the browser doesn't support frames, display this -->
   <noframes>
      <BODY>
      <p>
      This web site uses frames, but your browser doesn't
      support them.
      </p>
      </body>
   </noframes>
</frameset>
</html>
```

Figure 12.4 shows the result of this phase of completion, with the Banner page displayed in the banner frame. That's a start, but there's still a long way to go to have a truly active and exciting Web site! Continue on to the Contents page.

Note

You can find the files used to generate the version of the sample home page seen in Figure 12.4 on the companion CD-ROM in the SOURCE\CHAP14\STAGE2 directory.

Figure 12.4. The sample home page with the Banner page completed.

Stage 3: Developing the Contents Page (TOC.HTM)

The Contents page is useful because it allows you to get detailed about the contents of your Web site. Because this sample site only contains a few pages, the Contents page is pretty simple. Of course, you can use the ideas presented here to create much larger Contents pages. At this point, you need to know more about the content of pages that will appear in the topics frame.

Targeting the Contents of the Site

The Contents page will, of course, supply links to the Welcome, Topic 1, and Topic 2 pages, but it will also provide links to two anchors, or bookmarks, that you'll put in the Topic 2 page later in this chapter (in Stage 6). These anchors will be named, in the Topic 2 HTML source, as anchor1 and anchor2. Because you know this ahead of time, you can include links to the anchors on the Contents page now.

To include a link to an anchor, you use this syntax:

```
<A HREF="#AnchorName">
```

To include a link to a frame, you use this syntax:

```
<A HREF="filename.htm" TARGET="FrameName">
```

To include a link to an anchor that will appear in a frame, you use this syntax:

```
<A HREF="filename.htm#AnchorName" TARGET="FrameName">
```

Using these simple types of links, you can control the content within the topics frame from the contents frame. Listing 12.7 gives the complete HTML code for the Contents page (TOC.HTM).

Listing 12.7. The final HTML source code for the Contents page (TOC.HTM).

```
<HTML>

<HEAD>

<TITLE>Contents</TITLE>

</HEAD>

<BODY bgcolor="#FFFFF">

<H2><I>Contents</I></H2>

<HR>

<font face="Arial" size="2">

<!-- Welcome -->
```

```
<A HREF="welcome.htm" TARGET="topic" ALT="[Welcome]">
<IMG SRC="../gif/bullet2.gif" ALT="*" border="0">
<b>Welcome</b>
</a>
<p>
<!-- Topic 1 -->
<A HREF="topic1.htm" TARGET="topic" ALT="[Topic 1]">
<IMG SRC="../gif/bullet2.gif" ALT="*" border="0">
<b>Topic 1: A Java Applet</b>
</a>
<p>
<!-- Topic 2 -->
<A HREF="topic2.htm" TARGET="topic" ALT="[Topic 2]">
<IMG SRC="../gif/bullet2.gif" ALT="*" border="0">
<b>Topic 2: The Poetry Page</b>
</a>
<br>
<blockquote>
<li><A HREF="topic2.htm#anchor1" TARGET="topic" ALT="[Poem of the week]">
Poem of the Week<br>
</a></i>
<li><A HREF="topic2.htm#anchor2" TARGET="topic" ALT="[Runner up poem]">
Runner-Up
```

```
</a></li>

</blockquote>

<p>

</font>

</BODY>

</HTML>
```

Figure 12.5 shows the result of this phase of completion, with the Contents page displayed in the contents frame. Now you can move on to the Welcome page.

Note

The files used to generate the version of the sample home page seen in Figure 12.5 can be found on the companion CD-ROM in the SOURCE\CHAP14\STAGE3 directory.

Figure 12.5. The sample home page with the Contents page completed.

Stage 4: Developing the Welcome Page (WELCOME.HTM)

The Welcome page simply greets users and wows them with an ActiveX label control that expands and contracts as it changes colors. At the same time, background percussion music begins to play, and the famous Internet Explorer logo animated GIF spins around. Take a look at how all of this is accomplished.

Activating with ActiveX

The most prominent feature of this page is the animated label control. This animation is accomplished with VBScript and two ActiveX timer controls. To get the label control on the page, I used the ActiveX Control Pad, discussed in Chapter 11. After the control was in place, I added some VBScript to the page to control the label and the timers.

Controlling ActiveX with VBScript

The label control is named IeLabel1 and the two timer controls are named IETimer1 and IETimer2, respectively. One timer controls the size of the label, and the other timer controls the color. RGB color components are determined randomly by this VBScript function:

```
Function GetRandomColor(nMax)
   GetRandomColor = Int((nMax + 1) * Rnd)
End Function
```

The IETimer1_Timer() event controls the color changes, like this:

```
'
' Flashing text colors
'
Sub IeTimer1_Timer()
   crRed = GetRandomColor(256)
   crGreen = GetRandomColor(256)
   crBlue = GetRandomColor(256)
   IeLabel1.ForeColor = RGB(crRed, crGreen, crBlue)
end sub
```

The size of the label is determined by the VBScript nSize variable, like this:

```
'
' Resizing label
'
Sub IeTimer2_Timer()
   IeLabel1.FontSize = nSize
   if (nSize < 12) then
```

```
      nSize = 12
      bLarger = true
   elseif (nSize > 24) then
      nSize = 24
      bLarger = false
   end if
   if (bLarger) then
      nSize = nsize + 1
   else
      nSize = nsize - 1
   end if
end sub
```

Listing 12.8 gives the complete code listing for the Welcome page. Note the use of the <BGSOUND SRC> HTML tag, which plays a sound file when the page is displayed. In this case, the file is in MIDI format (although standard WAV files can be used as well). The background sound is specified like this:

```
<BGSOUND SRC="../media/percfun.mid">
```

This plays some percussive sounds that build in intensity the longer the page is displayed.

Listing 12.8. The final HTML source for the Welcome page (WELCOME.HTM).

```
<HTML>
<HEAD>
<TITLE>Welcome!</TITLE>
<META Name="keywords" Content="ActiveX Programming Unleashed">
</HEAD>
```

```
<BODY BACKGROUND="../gif/tan.gif">

<H2><i>Welcome!</i></H2>

<HR>

<P>

<IMG SRC="../gif/underc.gif" ALT="Under Construction!">

Welcome! This is just a sample Web site, but it utilizes a lot of

 cool technologies including: ActiveX, Java, JavaScript, VBScript,

 and more! It also uses state-of-the-art computer graphics,

 including ray-traced animation.

<p>

Browse what's here so far, and be sure to come back regularly!

 And remember:

<p>

<!--### Active "ActiveX Rocks!!" ###-->

<center>

<!-- MS Label Control -->

<OBJECT ID="IeLabel1" WIDTH=423 HEIGHT=49

 CLASSID="CLSID:99B42120-6EC7-11CF-A6C7-00AA00A47DD2"

 CODEBASE="http://www.microsoft.com/activex/controls/ielabel.ocx">

    <PARAM NAME="_ExtentX" VALUE="11192">

    <PARAM NAME="_ExtentY" VALUE="1296">

    <PARAM NAME="Caption" VALUE="ActiveX Rocks!!">

    <PARAM NAME="Angle" VALUE="0">

    <PARAM NAME="Alignment" VALUE="1">
```

```
    <PARAM NAME="Mode" VALUE="0">
    <PARAM NAME="FillStyle" VALUE="0">
    <PARAM NAME="FillStyle" VALUE="0">
    <PARAM NAME="ForeColor" VALUE="#000000">
    <PARAM NAME="BackColor" VALUE="#C0C0C0">
    <PARAM NAME="FontName" VALUE="Arial">
    <PARAM NAME="FontSize" VALUE="12">
    <PARAM NAME="FontItalic" VALUE="1">
    <PARAM NAME="FontBold" VALUE="1">
    <PARAM NAME="FontUnderline" VALUE="0">
    <PARAM NAME="FontStrikeout" VALUE="0">
    <PARAM NAME="TopPoints" VALUE="0">
    <PARAM NAME="BotPoints" VALUE="0">
<!-- For those browsers that can't see ActiveX... -->
<h2><i>ActiveX Rocks!! Get a new browser!</i></h2>.
</OBJECT>
</center>
<OBJECT ID="IeTimer1" WIDTH=39 HEIGHT=39
 CLASSID="CLSID:59CCB4A0-727D-11CF-AC36-00AA00A47DD2"
 CODEBASE="http://www.microsoft.com/activex/controls/ietimer.ocx">
    <PARAM NAME="_ExtentX" VALUE="1032">
    <PARAM NAME="_ExtentY" VALUE="1032">
    <PARAM NAME="Interval" VALUE="150">
```

```
</OBJECT>

<OBJECT ID="IeTimer2" WIDTH=39 HEIGHT=39

 CLASSID="CLSID:59CCB4A0-727D-11CF-AC36-00AA00A47DD2"

 CODEBASE="http://www.microsoft.com/activex/controls/ietimer.ocx">

    <PARAM NAME="_ExtentX" VALUE="1032">

    <PARAM NAME="_ExtentY" VALUE="1032">

    <PARAM NAME="Interval" VALUE="200">

</OBJECT>

<!-- OK, tie 'em together with some VBScript! -->

<SCRIPT LANGUAGE="VBScript">

<!--

dim crRed, crGreen, crBlue

dim nSize

dim bLarger

crRed = 255

crGreen = 230

crBlue = 0

nSize = 12

bLarger = true

Randomize

Function GetRandomColor(nMax)

   GetRandomColor = Int((nMax + 1) * Rnd)

End Function

'
```

```
' Flashing text colors
'
Sub IeTimer1_Timer()
   crRed = GetRandomColor(256)
   crGreen = GetRandomColor(256)
   crBlue = GetRandomColor(256)
   IeLabel1.ForeColor = RGB(crRed, crGreen, crBlue)
end sub
'
' Resizing label
'
Sub IeTimer2_Timer()
   IeLabel1.FontSize = nSize
   if (nSize < 12) then
      nSize = 12
      bLarger = true
   elseif (nSize > 24) then
      nSize = 24
      bLarger = false
   end if
   if (bLarger) then
      nSize = nsize + 1
   else
```

```
      nSize = nsize - 1
   end if
end sub
-->
</SCRIPT>
<!-- Display the IE3 animated gif -->
<CENTER>
<font face="Arial" size=-1 color="000000"><B>
<span style="font-size:9pt; color:000000">
Best experienced with<BR>
<A HREF="http://www.microsoft.com/ie/ie.htm">
<IMG SRC="../gif/ie3_anim.gif"
WIDTH=88 HEIGHT=31 ALT="Microsoft Internet Explorer"
   BORDER=0 ALIGN=center>
</A>
<br>Click here to start.
</span>
</font>
<!-- Keep the user apprised of last update -->
<H5>
<SCRIPT LANGUAGE="JavaScript">
<!-- Hide script from old browsers
   update = new Date(document.lastModified)
   month = update.getMonth() + 1
```

```
   date = update.getDate()
   year = update.getYear()
   document.writeln("<I>Last modified: " +
      month + "/" + date + "/" + year + "</I>")
   // End script hiding -->
</SCRIPT>
</H5>
</CENTER>
<p>
<!-- Some rhythmic background music -->
<BGSOUND SRC="../media/ percfun.mid">
</BODY>
</HTML>
```

Figure 12.6 shows the result of this phase of completion, with the Welcome page displayed in the topic frame.

Note

The files used to generate the version of the sample home page seen in Figure 12.6 can be found on the companion CD-ROM in the SOURCE\CHAP14\STAGE4 directory.

Figure 12.6. The final cut of the home page with WELCOME.HTM in the Topic pane.

Stage 5: The Topic 1 Page (TOPIC1.HTM)

Java applets provide yet another way to activate a Web site. The purpose of the Topic 1 page is to show off the use of a simple Java applet in a Web page. The applet is controlled by parameters given within the <APPLET> tag in the HTML source code. In this stage, I'll develop a simple Java applet called *SpinningRedTorus*. It's named this because the applet is an animation-only applet that displays a shiny red

torus spinning around its axis.

Note

For those of you not familiar with the term *torus*, it refers to a type of fourth-order, 3-dimensional mathematical surface called a quartic. All this lofty terminology really means is: a donut-shaped object. For instance, an inner tube is a torus.

Activating with Java

The Java programming language has taken the world by storm, and it was the initial impetus that began to bring Web pages to life. Java is a very intuitive and powerful object-oriented programming language with a well-defined class library. Java can create both applications, which run stand-alone, and applets, that run within a host application (such as a Java-enabled Web browser). Although Java is capable of creating much more than simple Web page applets, this discussion of Java will be restricted to this area of interest. Like ActiveX controls, Java applets can really spice up your Web pages!

Creating Java Applets with Visual J++

Sun Microsystems has licensed their Java technology to many software vendors, including Microsoft. Microsoft has released its Windows Java development system, Visual J++, to provide a familiar and comfortable working environment for Visual C++ users. The Visual J++ Developer Studio looks and acts exactly like the Visual C++ Developer Studio.

The Java Applet Wizard

You can easily create basic Java applets in Visual J++ by using the Applet Wizard, which walks you through several simple steps to define a skeletal Java applet. Figure 12.7 shows step 1 of the process, which allows you to define a class name for the applet, and to specify how the applet will execute (as an applet only, or as an applet and as an application).

Figure 12.7. Step 1 of the Java Applet Wizard.

Figure 12.8 shows step 2 of the Applet Wizard, which allows you to specify the width and height of the applet, and to specify whether a sample HTML file should be generated to test the compiled applet.

Figure 12.8. Step 2 of the Java Applet Wizard.

Figure 12.9 shows step 3 of the Applet Wizard, which allows you to create a multithreaded applet (or not)

and specify whether the applet will be used to display an animation. In this case, the SpinningRedTorus applet will be an animation-only applet. This step also allows you to specify whether the applet will respond to mouse messages (which SpinningRedTorus doesn't).

Figure 12.9. Step 3 of the Java Applet Wizard.

Defining Properties for the Applet

To control the rotation direction of the SpinningRedTorus (clockwise or counterclockwise) and the rotation velocity, two properties are needed: direction and interval. Step 4 of the Applet Wizard makes defining applet properties a snap, as you can see in Figure 12.10.

Figure 12.10. Step 4 of the Java Applet Wizard.

Generating the Applet Source Code

Once you've completed all the steps, Applet Wizard generates the source files for the applet, complete with default bitmaps that make up the animation frames. Figure 12.11 shows the SpinningRedTorus project loaded in the Microsoft Developer Studio.

Figure 12.11. The Developer Studio with the SpinningRedTorus project loaded.

Replacing the Default Bitmaps

The Applet Wizard uses 18 default bitmaps, depicting the earth rotating about its axis, but I replaced the default images with images from a 12-frame animation (of a spinning torus!) that I rendered using POVRAY.

The Applet Source Code

The applet source code is pretty straightforward and is well commented. When it executes in a browser, the applet loads the 12 images for its animation frames from the images directory, which *must* be a subdirectory of the applet's home directory. This images subdirectory is hard-coded into the applet, so SpinningRedTorus is very finicky about this location. The complete source code for the applet is given in Listing 12.9.

Listing 12.9. The complete source code for the SpinningRedTorus applet (SpinningRedTorus.java).

```
//************************************************************
// Applet   : SpinningRedTorus.java
//
// Author   : Rob McGregor
//
// Comments : Adapted from the Visual J++ Applet Wizard default
//            animation applet
//************************************************************
import java.applet.*;
import java.awt.*;
//============================================================
// Main Class for applet SpinningRedTorus
//============================================================
public class SpinningRedTorus extends Applet implements Runnable
{
   //----------------------------------------------------------
   // THREAD SUPPORT:
   //----------------------------------------------------------
   //   m_SpinningRedTorus Thread object for the applet
   //----------------------------------------------------------
   Thread    m_SpinningRedTorus = null;
   //----------------------------------------------------------
   // ANIMATION SUPPORT:
   //----------------------------------------------------------
```

```
// m_Graphics     stores the applet's Graphics context
// m_Images[]     the array of Image objects for the animation
// m_nCurrImage   the index of the next image to be displayed
// m_ImgWidth     width of each image
// m_ImgHeight    height of each image
// m_fAllLoaded   indicates whether all images have been loaded
// IMAGE_LAST     number of images used in the animation
//----------------------------------------------------------
private Graphics  m_Graphics;
private Image     m_Images[];
private int       m_nCurrImage;
private int       m_nImgWidth   = 0;
private int       m_nImgHeight  = 0;
private boolean   m_fAllLoaded  = false;
// Constants
private final int NUM_IMAGES = 12;
private final int FORWARD    = 1;
private final int BACKWARD   = 0;
//----------------------------------------------------------
// PARAMETER SUPPORT:
//----------------------------------------------------------
// Parameters allow an HTML author to pass information to
// the applet; the HTML author specifies them using the <PARAM>
```

```
  // tag within the <APPLET> tag. The following variables are
  // used to store the values of the parameters.
  //--------------------------------------------------------
  // Members for applet parameters
  private int m_Interval  = 100;
  private int m_Direction = 0;
  // Parameter names. To change a name of a parameter, you need
  // only make a single change. Simply modify the value of the
  // parameter string below.
  //--------------------------------------------------------
  private final String PARAM_Interval  = "Interval";
  private final String PARAM_Direction = "Direction";
  //========================================================
  // SpinningRedTorus Class Constructor
  //========================================================
  public SpinningRedTorus()
  {
     // Add constructor code here
  }
  //========================================================
  // public String getAppletInfo()
  //--------------------------------------------------------
  // Returns a string describing the applet's author, copyright
  // date, or miscellaneous information
```

```
//================================================================
public String getAppletInfo()
{
   return "Name: SpinningRedTorus\r\n" +
          "Author: Rob McGregor\r\n" +
          "Created with Microsoft Visual J++ Version 1.0";
}
//================================================================
// public String[][] getParameterInfo()
//----------------------------------------------------------------
// The getParameterInfo() method returns an array of strings
// describing the parameters understood by this applet.
//
// SpinningRedTorus Parameter Information:
//  { "Name", "Type", "Description" },
//================================================================
public String[][] getParameterInfo()
{
   String[][] info =
   {
      { PARAM_Interval, "int", "Interval between frames" },
      { PARAM_Direction, "int", "Forward or backward (1 or 0)" },
   };
```

```
    return info;
  }
  //===============================================================
  // public void init()
  //---------------------------------------------------------------
  // Called by the AWT when an applet is first loaded or reloaded
  // Override this method to perform whatever initialization your
  // applet needs, such as initializing data structures, loading
  // images or fonts, creating frame windows, setting the layout
  // manager, or adding UI components.
  //===============================================================
  public void init()
  {
     // The following code retrieves the value of each parameter
     // specified with the <PARAM> tag and stores it in a member
     // variable.
     //---------------------------------------------------------
     String param;
     // Interval: Interval between frames
     //---------------------------------------------------------
     param = getParameter(PARAM_Interval);
     if (param != null)
        m_Interval = Integer.parseInt(param);
     // Direction: Forward or backward (1 or 0)
```

```
    //---------------------------------------------------------
    param = getParameter(PARAM_Direction);
    if (param != null)
       m_Direction = Integer.parseInt(param);
    resize(160, 120);
  }
  //=========================================================
  // public void destroy()
  //---------------------------------------------------------
  // Called when the applet is terminating, place additional
  // applet clean up code here.
  //=========================================================
  public void destroy()
  {
     // Place applet cleanup code here
  }
  //=========================================================
  // public boolean imageUpdate()
  //---------------------------------------------------------
  // The imageUpdate() method is called repeatedly by the AWT
  // while images are being constructed.  The flags parameter
  // provides information about the status of images under
  // construction. This method checks whether the ALLBITS flag is
```

```
  // set, which means that an image is completely loaded. When all
  // the images are completely loaded, this method sets the
  // m_fAllLoaded variable to true so that animation can begin.
  //================================================================
  public boolean imageUpdate(Image img, int flags, int x, int y,
     int w, int h)
  {
     // Nothing to do if images are all loaded
     //----------------------------------------------------------
     if (m_fAllLoaded)
        return false;
     // Want all bits to be available before painting
     //----------------------------------------------------------
     if ((flags & ALLBITS) == 0)
        return true;
     // All bits are available, so increment loaded count of fully
     // loaded images, starting animation if all images are loaded
     //----------------------------------------------------------
     if (++m_nCurrImage == NUM_IMAGES)
     {
        m_nCurrImage =
           (FORWARD == m_Direction) ? 0 : NUM_IMAGES - 1;
        m_fAllLoaded = true;
     }
```

```
    return false;
  }
  //===============================================================
  // private void displayImage()
  //---------------------------------------------------------------
  // Draws the next image (if all images are currently loaded)
  //===============================================================
  private void displayImage(Graphics g)
  {
     if (!m_fAllLoaded)
        return;
     // Draw Image in center of applet
     //---------------------------------------------------------
     g.drawImage(m_Images[m_nCurrImage],
        (size().width - m_nImgWidth) / 2,
        (size().height - m_nImgHeight) / 2, null);
  }
  //===============================================================
  // public void paint(Graphics g)
  //---------------------------------------------------------------
  // SpinningRedTorus Paint Handler
  //===============================================================
  public void paint(Graphics g)
```

```
  {
     // The following code displays a status message until all the
     // images are loaded. Then it calls displayImage to display
     // the current image.
     //-----------------------------------------------------------
     if (m_fAllLoaded)
     {
        Rectangle r = g.getClipRect();


        g.clearRect(r.x, r.y, r.width, r.height);
        displayImage;
     }
     else
        g.drawString("Loading ray traced images...", 10, 20);
  }
  //============================================================
  // public void start()
  //-----------------------------------------------------------
  // Called when the page containing the applet first appears on
  // the screen to start execution of the applet's thread.
  //============================================================
  public void start()
  {
     if (m_SpinningRedTorus == null)
```

```
    {
       m_SpinningRedTorus = new Thread(this);
       m_SpinningRedTorus.start();
    }
  }

  //=============================================================
  // public void stop()
  //-------------------------------------------------------------
  // Called when the page containing the applet is no longer on
  // the screen. This method stops execution of the applet's
  // thread.
  //=============================================================
  public void stop()
  {
    if (m_SpinningRedTorus != null)
    {
       m_SpinningRedTorus.stop();
       m_SpinningRedTorus = null;
    }
  }
  //=============================================================
  // public void run()
```

```
  //---------------------------------------------------------------
  // Called when the applet's thread is started. If your applet
  // performs any ongoing activities without waiting for user
  // input, the code for implementing that behavior typically goes
  // here. For example, for an applet that performs animation,
  // the run() method controls the display of images.
  //===============================================================

  public void run()
  {
     repaint();
     m_Graphics   = getGraphics();
     m_nCurrImage = 0;
     m_Images     = new Image[NUM_IMAGES];
     // Load in all the images
     //---------------------------------------------------------
     String strImage;
     // For each image in the animation, this method first
     // constructs a string containing the path to the image file;
     // then it begins loading the image into the m_Images array.
     // Note that the call to getImage() will return before the
     // image is completely loaded.
     //---------------------------------------------------------
     for (int i = 1; i <= NUM_IMAGES; i++)
```

```
    {
       // Build path to next image
       //-----------------------------------------------------
       strImage = "./images/img00" + ((i < 10) ? "0" : "")
          + i + ".gif";
       m_Images[i-1] = getImage(getDocumentBase(), strImage);
       // Get width and height of one image.
       // Assuming all images are same width and height
       //-----------------------------------------------------
       if (m_nImgWidth == 0)
       {
          try
          {
             // The getWidth() and getHeight() methods of the
             // Image class return -1 if the dimensions are not
             // yet known. The following code keeps calling
             // getWidth() and getHeight() until they return
             // actual values.
             //
             // NOTE:
             // This is only executed once in this loop, since
             // we are assuming all images are the same width
             // and height.  However, since we do not want to
```

```
            // duplicate the above image load code, the code
            // resides in the loop.
            //-------------------------------------------------
            while ((m_nImgWidth =
               m_Images[i-1].getWidth(null)) < 0)
               Thread.sleep(1);
            while ((m_nImgHeight =
               m_Images[i-1].getHeight(null)) < 0)
               Thread.sleep(1);
         }
         catch (InterruptedException e)
         {
            // Place exception-handling code here in case an
            // InterruptedException is thrown by Thread.sleep(),
            // meaning that another thread has interrupted this
            // one
         }
      }
      // Force image to fully load
      //-----------------------------------------------------
      m_Graphics.drawImage(m_Images[i-1], -1000, -1000, this);
   }
   // Wait until all images are fully loaded
   //-------------------------------------------------------
```

```
while (!m_fAllLoaded)
{
   try
   {
      Thread.sleep(10);
   }
   catch (InterruptedException e)
   {
      // Place exception-handling code here in case an
      // InterruptedException is thrown by Thread.sleep(),
      // meaning that another thread has interrupted this
      // one
   }
}

repaint();
while (true)
{
   try
   {
      // Draw next image in animation
      //-----------------------------------------------------
      displayImage(m_Graphics);
```

```
            if (m_Direction == FORWARD)   // 1
            {
               m_nCurrImage++;
               if (m_nCurrImage == NUM_IMAGES)
                  m_nCurrImage = 0;
            }
            if (m_Direction == BACKWARD)  // 0
            {
               m_nCurrImage--;
               if (m_nCurrImage <= 0)
                  m_nCurrImage = NUM_IMAGES - 1;
            }
            // Set the animation to desired speed
            Thread.sleep(m_Interval);
         }
         catch (InterruptedException e)
         {
            // Place exception-handling code here in case an
            // InterruptedException is thrown by Thread.sleep(),
            // meaning that another thread has interrupted this
            // one
            stop();
         }
      }
```

```
   }

}
```

Adding the Applet to the Web Site

Now that you have a custom Java applet on your hands, how do you add it to a Web page? Simple! Using the HTML <APPLET> tag, you can define the applet for Java-enabled browsers by using the following syntax:

```
<applet

   code   = [AppletClass].class

   id     = [AppletID]

   width  = [AppletWidth]

   height = [AppletHeight]>

   <param name=[Param1Name] value=[Value]>

   <param name=[Param2Name] value=[Value]>

   ...etc...

</applet>
```

Recall that the SpinningRedTorus applet uses two parameters, Direction and Interval. Interval specifies the milliseconds paused between frames, and Direction specifies either clockwise (1) or counterclockwise (0) rotation. The Topic 1 page is the lucky recipient of the SpinningRedTorus applet and uses parameters of Interval=100, and Direction=1 (clockwise rotation), like this:

```
<applet

   code   = SpinningRedTorus.class

   id     = SpinningRedTorus

   width  = 160

   height = 120>
```

```
   <param name=Interval value=100>

   <param name=Direction value=1>

</applet>
```

Take a look at Listing 12.10, which shows the complete source code for TOPIC1.HTM. Note that a background music file is specified by this line:

```
<BGSOUND SRC="../media/ludwig5.mid">
```

This is a MIDI file that plays the opening of Beethoven's Fifth Symphony (it seemed a fitting addition to the drama of the spinning torus).

Listing 12.10. The final HTML source code for the Topic 1 page (TOPIC1.HTM).

```
<html>

<head>

<title>Topic 1: A Spinning Red Torus</title>

</head>

<body bgcolor="#FFFFF">

<!-- Some not so soothing background music -->

<BGSOUND SRC="../media/ludwig5.mid">

<h2><i>

Topic 1: A Java Applet

</i></h2>

<hr>

<h3>Rob's Ray-Traced Shiny-Red Torus and Gold Sphere Java

 Applet...</h3><p>

<h4>This 'lil Java applet is 160x120 pixels in size and takes
```

```
 two parameters: </h4><p>
<IMG SRC="../gif/bullet2.gif" ALT="*">
<b><i>Interval</i></b> specifies the milliseconds paused between
 frames, and <br>
<IMG SRC="../gif/bullet2.gif" ALT="*">
<b><i>Direction</i></b> specifies either clockwise (1) or
 counterclockwise (0) rotation.<p>
For this example page, the <tt><b>Interval=100</b></tt>,
 and the <tt><b>Direction=1</b></tt> (clockwise):
<p>
<applet
    code=SpinningRedTorus.class
    id=SpinningRedTorus
    width=160
    height=120 >
    <param name=Interval value=100>
    <param name=Direction value=1>
</applet>
<hr>
Check out the <a HREF="../java/SpinningRedTorus.java"
TARGET="topic" BORDER="0" ALT="Topic 1">
<b>source code </b></a>
for this applet by clicking the link provided. If you have
 Visual J++ installed on your system, it will load the applet
```

```
 project!

</body>

</html>
```

Figure 12.12 shows the result of this stage of completion, with the Topic 2 page displayed in the topic frame. Now on to the final page.

Note

The files used to generate the version of the sample home page seen in Figure 12.12 can be found on the companion CD-ROM in the SOURCE\CHAP14\STAGE5 directory.

Figure 12.12. The sample home page with the Applet Page completed.

Stage 6: The Topic 2 Page (TOPIC2.HTM)

This final stage of the sample Web site simply shows how to add some anchors, or bookmarks, to a Web page. You'll create the two anchors mentioned back in stage 3, when you developed the Contents page, and add a few images to the page to make it look nice.

Defining the Anchors

Recall from the HTML code in TOC.HTM that you are to name the two anchors on this page anchor1 and anchor2. This is a simple process, using the <A NAME="*Name*"> syntax to define an anchor. The first anchor is then defined as:

```
<a name="anchor1"><h3>Poem of The Week</h3></a>
```

The second anchor is defined as:

```
<a name="anchor2"><h3>Runner Up</h3></a>
```

Adding a Marquee

To add a little activity to the Topic 2 page, I added a marquee, which scrolls text across the screen. Although the marquee is a bit cliched because of its overuse on the Web, it does help to bring this otherwise static page to life. The marquee uses the HTML <MARQUEE> tag and is defined like this:

```
<FONT SIZE=4 COLOR=#FF0000>

<MARQUEE BEHAVIOR=SLIDE DIRECTION=LEFT BORDER="0">

Poems of the week...

</MARQUEE>

</FONT>
```

This sets the font size and color, and it tells the browser to start displaying the text beyond the right edge of the window and slide the text to the left.

For cosmetic reasons, the background color of this page is set to a subdued aqua color with the BGCOLOR attribute set in the <BODY> tag, like this:

```
<BODY BGCOLOR="#92DDD4">
```

The complete HTML source code for this final page of the sample Web site is shown in Listing 12.11.

Listing 12.11. The HTML source code for the final version of the Topic 2 page (TOPIC2.HTM).

```
<HTML>

<HEAD>

<TITLE>Topic 2: The Poetry Corner</TITLE>

</HEAD>

<BODY BGCOLOR="#92DDD4">

<H2><I>Topic 2: The Poetry Corner</I></H2>

<p>
```

```
<!-- A cliche marquee -->
<FONT SIZE=4 COLOR=#FF0000>
<MARQUEE BEHAVIOR=SLIDE DIRECTION=LEFT BORDER="0">
Poems of the week...
</MARQUEE>
</FONT>
<hr>
Hello, and welcome to the poetry corner, where each and every week
 you can feed your soul! This time around the poem of the week is
 one of my favorites: "The Road Not Taken"
<p>
<center>
<a name="anchor1"><h3>Poem of The Week</h3></a>
<img src="../gif/frost1.gif" align=bottom width=302 height=450>
</center>
<hr>
We've also got a poem this week from one of Florida's most
 talented young poets. The piece is entitled, "Yesterday"
<p>
<center>
<a name="anchor2"><h3 align>Runner Up</h3></a>
<img src="../gif/yester.gif" align=bottom
   width=251 height=453>
</center>
```

```
</BODY>

</HTML>
```

Figure 12.13 shows the result of this final stage, with the Topic 2 page displayed in the topic frame.

Note

The files used to generate the version of the sample home page seen in Figure 12.13 can be found on the companion CD-ROM in the SOURCE\CHAP14\STAGE6 directory.

Figure 12.13. The sample home page with the Topic 2 displayed in the Topic frame.

Summary

Creating exciting, interactive Web pages and Web sites is both challenging and fulfilling. It takes both artistic and programming talent, as well as an eye for detail. This chapter has shown you how to create a basic frames-driven Web site that contains six HTML files. Each of these six files demonstrated a different aspect of Web page design, including the use of: frames, ActiveX controls with VBScript, Java applets, JavaScript, image and text links to other frames, and background sounds, among other things.

Now that you've seen how the client side works, it's time to explore the server side of things. In the next chapter, you'll learn how to get greater control over your Web sites with Windows CGI Scripting.

Chapter 13

Windows CGI Scripting

by Wayne Berry

Common Gateway Interface (CGI) was developed so that a Web browser could pass parameters to a Web server, regardless of the platform on which either of the machines is running. For instance, if the Web browser is running on a Macintosh and the server is running on a UNIX machine, the browser could pass parameters that both machines can understand. *CGI scripts* is a common name for programs that run on the server side. The name *scripts* comes from the fact that in the beginning, most server-side implementations were Perl scripts running on a UNIX box. CGI scripts do not have to be Perl scripts; they can be any program, scripted or compiled, that the Web server is allowed to execute.

CGI Scripting—the Client Side

The parameters to the CGI scripts usually originate in a <FORM></FORM> tag within the current page that the browser is displaying. Encapsulated within the <FORM> tag are <INPUT> tags that create 'name = value' parameters. There are also other HTML tags that create 'name = value' parameter pairs. When the form is submitted to the server, the browser encodes the parameters into the CGI standard. The browser takes all the parameters and puts them into a single string without spaces. The browser replaces the spaces with a + symbol and converts all other symbols to hexadecimal values preceded with a % symbol. The first half of the string is the location of the URL on the server that contains the location of the CGI script. The second half of the string is 'name = value' pairs. The two halves are separated by a ? symbol and the 'name = value' pairs are separated by a & symbol.

Table 13.1 is an example of the HTML Input

Table 13.1. HTML to CGI.

HTML Input CGI <INPUT NAME="Item" VALUE="1"> Item=1 <INPUT NAME="FirstName" VALUE="John Doe"> FirstName=John+Doe <INPUT NAME="Company" VALUE="Al's RVs"> Company=Al%27s+RVs

Simple Form Examples

Here are several examples of HTML text that can be used to pass parameters to a CGI Script.

If the browser is reading a single input form page that contains these tags:

```
<FORM ACTION="http://www.myserver.com/scripts/mycgi.exe">

<INPUT TYPE=HIDDEN NAME="Item" VALUE="Oranges">

<INPUT TYPE=SUBMIT>

</FORM>
```

the server would receive a request of

```
http:\\www.myserver.com\scripts\mycgi.exe?Item=Oranges
```

If the browser is reading a multiple input form page that contains these tags

```
<FORM ACTION="http://www.myserver.com/scripts/mycgi.exe">

<INPUT TYPE=HIDDEN NAME="Item" VALUE="Oranges">
```

```
<INPUT TYPE=HIDDEN NAME="Price" VALUE="2.00">

<INPUT TYPE=SUBMIT>

</FORM>
```

the server would receive a request of

```
http://www.myserver.com/scripts/mycgi.exe?Item=Oranges&Price=2.00
```

Hidden INPUT types like those in the preceding are not shown to the user by the browser. More interesting INPUT types and other tags allow user interaction. If the browser is reading a page that contains these tags (where the INPUT is associated with a text box, and the SELECT is associated with a drop-down list containing two choices: 1.00 or 2.00), you would have code something like this:

```
<FORM ACTION="http://www.myserver.com/scripts/mycgi.exe">

<INPUT TYPE=TEXT NAME="Item">

<SELECT NAME="Price"'>

<OPTION>1.00

<OPTION>2.00

</SELECT>

<INPUT TYPE=SUBMIT>

</FORM>
```

When the user selects the link, the parameters will be sent to mycgi.exe just like the multiple input form's response..

CGI Scripting—the Server Side

On receiving a request from a browser, the Web server interrupts the request and decides if it is a request for a CGI script or a static page, such as an HTML text file. If the request is for a CGI script, the server passes the parameters that are referenced in the URL to the CGI script. The CGI script decodes the parameters and uses them to create output that is sent back to the server. The server takes the output and returns it to the browser.

What Makes Up an HTML Page?

As a static text file, HTML pages start with the <HTML> tag and end with the </HTML> tag. When the server reads a static page from the disk, it outputs not only the text file to the browser, but also adds a header at the beginning. The header contains information for the browser that describes the server state and the content of the information following the header. The header contains a status line to indicate that the server transaction completed successfully, and it also contains a line to indicate the format of the information following the header.

When your CGI script executes, it must generate its own header information to be sent back to the server. The capability of sending back the header line allows the script to notify the server whether or not it has run successfully. For now, assume that the CGI script is returning as an HTML page to the browser.

A successful execution written in the header might look like this:

```
Status: 200

Content-type: text/html

<HTML>

<BODY>

Hello World

</BODY>

</HTML>
```

Notice the extra line between the header and the first <HTML> tag. This is very important syntax for the Web browser; without this line, your CGI script won't run properly. The server returns the output of your CGI script to the client's browser. When viewed as source from the browser, the header won't be displayed. 200 is a standard success code. Table 13.2. contains other code.

Table 13.2. Header Return Codes and their meanings

Return Code Meaning 200 Request completed 201 Object created, reason = new URI 202 Asynchronous completion (TBS) 203 Partial completion 204 No information to return 300 Server couldn't decide what to return 301 Object permanently moved 302 Object temporarily moved 303 Redirection with new access method 304 If-modified-since was not modified 400 Invalid syntax 401 Access denied 402 Payment required 403 Request forbidden 404 Object not found 405 Method is not allowed 406 No response acceptable to client found 407 Proxy authentication required 408 Server timed out waiting for request 409 User should resubmit with more information 410 The resource is no longer available 411 Couldn't authorize client 500 Internal server error 501 Required not supported 502 Error response received from gateway 503 Temporarily overloaded 504 Timed out waiting for gateway

Notice that codes in the 200s are used when the action was successfully received, understood, and accepted. Codes in the 300s are used when further action must be taken in order to complete the request. Codes in the 400s are used when the request contains bad syntax or cannot be fulfilled. Codes in the 500s are used when the server failed to fulfill an apparently valid request.

When your CGI script runs into an error, either processing information or accessing resources like SQL Server, it should return both an error status code and some HTML text describing the problem. For instance:

```
Status: 500

Content-type: text/html

<HTML>

<BODY>

Server Error, please try again later

</BODY>

</HTML>
```

The Development Environment

Before you get started making your own scripts and viewing them, you need to have a development environment. I prefer to have a single machine running NT 3.51 that has both the browser, the Web server, and my compiler on it. Debugging takes place on the Web server because the CGI scripts are running on the Web server. With Microsoft Developer Studio, you can debug the CGI scripts you write, so it makes sense to have both the compiler and the Web server on the same machine. I prefer not to swivel between two machines because it will be the browser that activates your scripts on the Web server. Finally, you need to have a Web server in which there is low activity. A poorly written script can crash the Web server, causing downtime for other users.

A Word About Security

Writing CGI scripts for Web servers is like allowing everyone to run a program on your machine. The first step to good security is to make sure that the users cannot read your script. This isn't a problem if you use compiled program written in a language such as C++. This is a concern when you're writing in batch or another type of runtime script. Make sure that the directory that contains your script has EXECUTE permissions, but not READ. A good example is the default IIS (Microsoft's Internet Information Server) script directory. This will allow users to execute the CGI scripts but not read them.

Using Batch Files

The default installation of IIS can execute batch files as CGI scripts. Batch files, considered the scripting language of DOS, are not as powerful as Perl scripts in UNIX. Because batch files lack string handling functions, they are limited in use as CGI scripts. Their only advantage is that they are a runtime language and make good sample programs.

Hello World!

Let's create a "Hello World!" CGI script. Create a batch file named List13_1.bat in the scripts directory of your Web server, as shown in Listing 13.1.

Listing 13.1. The Hello World Example

```
@echo off

REM Header

echo Status: 200

echo Content-type: text/html

echo.

REM Body

echo "<HTML><BODY>Hello World!</BODY></HMTL>"
```

To run the script, type a URL address into your browser as *MYMACHINE*\Scripts\Lst13_1.bat, where *MYMACHINE* is the name of your computer.

The first thing to notice is that "Hello World!" is in quotes. This is because the echo statement thinks that > is the symbol to pipe the output to a device. In double quotes, the > is outputted instead of the piped. This problem is the reason that batch files make poor CGI scripts.

How the CGI Script Accesses the Parameters

The server puts the CGI parameters into the environment variable QUERY_STRING. QUERY_STRING contains the CGI string in its CGI-coded form. It is the responsibility of the CGI script to get the information it needs from QUERY_STRING. Unfortunately, batch files do not have functions for string manipulation. This means that you will be able to view the CGI string but not separate the '*name* = *value*' pairs or translate the CGI hexadecimal values. This problem is another good reason not to use batch files for CGI scripts.

Listing 13.2 allows the user to view the CGI parameters passed to the batch file.

Listing 13.2. A batch file that returns the Query string.

```
@echo off

REM Header

echo Status: 200
```

```
echo Content-type: text/html

echo.

REM Body

echo "<HTML><BODY>Query String: %QUERY_STRING%</BODY></HMTL>"
```

To run the script, type a URL address into your browser as *MYMACHINE*\Scripts\Lst13_2.bat, where *MYMACHINE* is the name of your computer. Notice that the QUERY_STRING doesn't contain a value; the browser displays "Query String:" as the text on the HTML page. Now change the URL to *MYMACHINE*\Scripts\Lst13_2.bat?Name=John. The browser now displays "Query String: Name=John" as the text on the HTML page.

Other Parameters Passed to the Server

Besides the parameters passed to the server as part of the URL, the server also puts other information about the browser and the server state in environment variables (see Table 13.3).

Table 13.3. Server parameters.

HTTP variable Meaning HTTP_USER_AGENT The maker and version of the browser. Best used for making adjustments in the layout for different browsers. HTTP_REFERER The URL that the client is referencing. HTTP_CONTENT_TYPE The Content type of the input passed to the page. HTTP_CONTENT_LENGTH The length of the content passed to the page. CONTENT_LENGTH The length of the content passed to the page. CONTENT_TYPE The Content type of the input passed to the page. GATEWAY_INTERFACE Version of CGI handled by the server. HTTP_ACCEPT The type of input the browser will expect. PATH_INFO The pathname of the URL without the server name. PATH_TRANSLATED The full path to the CGI script. This variable is great for getting a location where you can write additional files. REMOTE_ADDR The IP address of the client. REMOTE_HOST The hostname of the client. REMOTE_USER This contains the username supplied by the client and authenticated by the Server. SCRIPT_NAME The name of the script being executed. SERVER_NAME The name of the server. This is good for referencing other pages on this server. SERVER_PROTOCOL The version of the HTML protocol the server is using. SERVER_SOFTWARE The Web server name and version. REQUEST_METHOD The type of request used either GET or POST. QUERY_STRING The CGI string passed with a GET Request.

Listing 13.3 is an example of a batch file that displays all the major environment variables.

Listing 13.3. Views all the return values from a batch file CGI script.

```
@echo off

REM Header

echo Status: 200
```

```
echo Content-type: text/html

echo.

REM Body

echo "<HTML><BODY>"

echo QUERY_STRING: %QUERY_STRING% "<BR>"

echo ALL_HTTP: %ALL_HTTP% "<BR>"

echo HTTP_USER_AGENT: %HTTP_USER_AGENT% "<BR>"

echo HTTP_REFERER: %HTTP_REFERER% "<BR>"

echo HTTP_CONTENT_TYPE: %HTTP_CONTENT_TYPE% "<BR>"

echo HTTP_CONTENT_LENGTH: %HTTP_CONTENT_LENGTH% "<BR>"

echo HTTP_EXTENSION: %HTTP_EXTENSION% "<BR>"

echo AUTH_TYPE: %AUTH_TYPE% "<BR>"

echo CONTENT_LENGTH: %CONTENT_LENGTH% "<BR>"

echo CONTENT_TYPE: %CONTENT_TYPE% "<BR>"

echo GATEWAY_INTERFACE: %GATEWAY_INTERFACE% "<BR>"

echo HTTP_ACCEPT: %HTTP_ACCEPT% "<BR>"

echo PATH_INFO: %PATH_INFO% "<BR>"

echo PATH_TRANSLATED: %PATH_TRANSLATED% "<BR>"

echo REMOTE_ADDR: %REMOTE_ADDR% "<BR>"

echo REMOTE_HOST: %REMOTE_HOST% "<BR>"

echo REMOTE_USER: %REMOTE_USER% "<BR>"

echo REQUEST_METHOD: %REQUEST_METHOD% "<BR>"

echo SCRIPT_NAME: %SCRIPT_NAME% "<BR>"

echo SERVER_NAME: %SERVER_NAME% "<BR>"

echo SERVER_PORT: %SERVER_PORT% "<BR>"
```

```
echo SERVER_PROTOCOL: %SERVER_PROTOCOL% "<BR>"

echo SERVER_SOFTWARE: %SERVER_SOFTWARE% "<BR>"

echo "</BODY></HMTL>"
```

Save the preceding example as Lst13_3.bat in your scripts directory and call it from your wwwroot directory by creating a Lst13_3.htm that looks like this:

```
<HTML>

<BODY>

<FORM ACTION="http://MYMACHINE/scripts/lst13_3.bat">

Name: <INPUT TYPE=TEXT NAME="Name">

<INPUT TYPE=SUBMIT>

</FORM>

</BODY>

</HTML>
```

The Difference Between *POST* and *GET*

There are two TYPE methods that FORM can use to transmit the parameters for the CGI script: GET and POST. You can choose which method to use in the <FORM> tag by entering METHOD=POST or METHOD=GET. By not entering any action, the form defaults to GET. Both methods send information by the way of CGI to the server, but in different ways.

The main difference between POST and GET is the way in which you receive the CGI parameters. With GET, you get the parameters through QUERY_STRING. With POST, the parameters are piped into the batch file through standard input (stdin). Another difference is that GET can support only 255 characters in the CGI string. POST has an unlimited number.

Change Lst13_3.htm's FORM to read

```
<FORM ACTION="http://MYMACHINE/scripts/Lst13_7.bat" METHOD=POST>
```

Now, try resubmitting the CGI script to the Lst13_3.bat. Notice that the QUERY_STRING isn't filled in. Also notice that CONTENT_LENGTH has a value of 10 with the GET method and a value of 0 with the POST method. Look at the address space of the URL at the top of your browser: with a GET request the CGI script parameters

appeared, but with a POST request the CGI script parameters don't appear. This is important for passing confidential information from one page to another. POST also gives a cleaner look to your Web page. Finally, make note that the REQUEST_METHOD is POST and not GET. The differences between POST and GET are listed in Table 13.4.

Table 13.4. Differences between POST and GET.

POST GET Doesn't display parameters Displays parameters in the address line
in the address line of the of the URL
URL Doesn't set QUERY_STRING Sets QUERY_STRING Sets CONTENT_LENGHT Doesn't set
CONTENT_LENGTH Unlimited CGI string length Limited to 255 characters in CGI string

Batch files have no way of supporting standard input (stdin). By using CGI scripts created in C++, you can handle standard in and separate the '*name* = *value*' pairs passed in by the form.

Hello World in C++

To create a C++ CGI script, I use VC++ 4.1 and MFC. All C++ CGI scripts must be console applications, in Microsoft Windows. It's important to remember that there is a possibility of more than one user using your application at a time. For every user executing the script, a new instance of the application will be open.

To create a C++ CGI script start by

1. 1. Open Microsoft Developer Studio.
2. 2. Choose File | New.
3. 3. Choose Project Workspace from the New list box and click OK.
4. 4. From the Type list box choose Console Application.
5. 5. Name the project Lst13_4.
6. 6. Click Create.
7. 7. You now need to include MFC. Select Build from the menu bar, then select Settings. In the General Tab, choose "Use MFC in a Shared Dll (mfc40.dll)" from the Microsoft Foundation Classes select Box. Click OK.
8. 8. Open a blank text file by selecting File from the Menu bar, clicking on New, and then selecting text file.
9. 9. Save the file as Lst13_4.cpp. Select File|Save. Call the file Lst13_4.cpp.
10. 10. All that is left is to include the code file in the project. Select Insert from the menu bar and then select Insert Files into Project. Select Lst13_4.cpp and then press Add.
11. Insert the text in Listing 13.4 into Lst13_4.cpp.

Listing 13.4. A C++ Hello World Example

```
// lst13_4.cpp

#include <stdio.h>

void main( int argc, char *argv[ ], char *envp[ ] )

{
```

```
// Header

printf("Status: 200\r\n");

printf("Content-type: text/html\r\n");

printf("\r\n");

// Body

printf("<HTML><BODY>Hello World!</BODY></HTML>\n");

}
```

1. 11. Now compile the project. Make sure the configuration selected is Lst13_4—Win32 Debug. Select Build Settings from the Menu bar and then select Build Lst13_4.exe.
2. 12. Copy Lst13_4.exe from the debug directory to the scripts directory of your IIS.
3. 13. To View your CGI script, open your browser and enter this line as the URL: *MYMACHINE*\Scripts\Lst13_4.exe, where *MYMACHINE* is the name of your computer.

In step 7, you chose a shared mfc40.dll instead of a static library. The reason for this is that it cuts down on operating overhead. To reduce operating overhead, having a smaller executable is better. Sharing DLLs makes for smaller executables because the MFC code is in a DLL, not bound to your executable. If more than one person is using your application, then more than one of your applications is loaded into memory, but only one shared mfc40.dll is loaded. Also, the smaller your CGI script, the quicker it will load, allowing the page to be sent to the user faster. The thing to remember about using shared DLLs is that you will need to copy mfc.dll to the server along with your CGI script. Remember that the preceding example assumes that your compiler and your server are on the same machine. This means that mfc.dll will be in your system path and you will not need to copy it.

In step 7, you chose to use MFC, yet the code used as an example didn't reference MFC. The preceding example is intended to be used as a generic example of how to create a CGI script. Other examples in this chapter will reference MFC.

Make sure to compile the debug configuration. CGI Scripts will run in both debug and release. For debugging purposes, you need to have a debug build. Final products should be in a release build because release executables are smaller and take less time for the Web server to load.

More advanced users of Microsoft Developer Studio may want to create their projects within the scripts directory of the IIS. The advantage of this is that you don't have to copy the executable (Step 12). When naming the project, select the scripts directory as the location instead of the default projects directory. When you compile your executable, the CGI script will be built into the scripts directory. The browser URL will also be different:

```
MYMACHINE/scripts/Lst13_4/debug/Lst13_4.exe
```

where *MYMACHINE* is the name of your computer.

The CGI scripts are no different then a regular console application. They can be run from a DOS command line and will display exactly the same information that it sends to the server. In fact, the way to send information to the server is to write to standard out (stdout), just like a console application. Running the console application is a way of debugging your executable.

Notice the header section of example nine source code. Each header line has both a new line character \n and a carriage return \r; these are required. Also note that the required space is represented by a \r\n.

Getting the CGI Parameters in C++

Create another console application as you did in Listing 13.4, call it Lst13_5, and use the source from Listing 13.5. Compile it and copy it to your server's scripts directory.

Listing 13.5. An example of viewing the query string.

```
// lst13_5.cpp

#include <stdio.h>

#include <afx.h>

void main( int argc, char *argv[ ], char *envp[ ] )

{

// Header

printf(_T("Status: 200\r\n"));

printf(_T("Content-type: text/html\r\n"));

printf(_T("\r\n"));

// Body

printf(_T("<HTML><BODY>"));

DWORD  dwBufferSize=50;

LPTSTR szQuery = new TCHAR[dwBufferSize];

GetEnvironmentVariable(_T("QUERY_STRING"),szQuery, dwBufferSize);

printf(_T("QueryString: %s"),szQuery);

printf(_T("</BODY></HTML>\n"));

delete szQuery;

}
```

Copy Listing 13.3's HTML form from Lst13_3.htm to Lst13_5.htm and change the Form attributes to read

```
<FORM ACTION="http://MYMACHINE/scripts/Lst13_5.exe" METHOD=GET>
```

Load Lst13_5.htm in your browser, type a name, and submit the data to your C++ CGI script. The query string should represent the name you typed.

With the power of C++, you can take the query string passed by the server and resolve the '*name* = *value*' pairs of CGI into C variables that you can use. In addition, you can retrieve the information from standard input (stdin) and process post methods.

First, you must make a form that sends interesting data to your CGI script. Save the code in Listing 13.6 as lst13_6.htm in the wwwroot directory of your server.

Listing 13.6. Reading CGI parameters.

```
<HTML>

<BODY>

<FORM ACTION="http://MYMACHINE/scripts/Lst13_10.exe" METHOD=GET>

Name: <INPUT TYPE=TEXT NAME="Name"><BR>

Age: <INPUT TYPE=TEXT NAME="Age"><BR>

<INPUT TYPE=SUBMIT>

</FORM>

</BODY>

</HTML>
```

Create, compile, and copy to the server a CGI script called Lst13_6.exe that contains the following source code:

```
// Lst13_6.cpp

#include <stdio.h>

#include <afx.h>

// Global Cache for Post

// You can only read from

// Standard In Once

TCHAR *szCache=NULL;
```

```
TCHAR ConvertHex(TCHAR cHigh, TCHAR cLow)
{
static const TCHAR szHex[] = _T("0123456789ABCDEF");
LPCTSTR pszLow;
LPCTSTR pszHigh;
TCHAR cValue;
// Find the Values in the Hex String
pszHigh = _tcschr(szHex, (TCHAR) _totupper(cHigh));
pszLow = _tcschr(szHex, (TCHAR) _totupper(cLow));
// If both Values Exist Then Calculate the Value
// Based off of the string
if (pszHigh && pszLow)
{
cValue = (TCHAR) (((pszHigh - szHex) << 4) + (pszLow - szHex));
return (cValue);
}
return('?');
}
// Returns the String
LPVOID TranslateCGI(LPTSTR pszString)
{
LPTSTR pszIndex = pszString;
LPTSTR pszReturn = pszString;
// unescape special characters
while (*pszIndex)
```

```
{
// Translate '+' to Spaces
if (*pszIndex == _T('+'))
*pszReturn++ = _T(' ');
// Translate Hex Strings to characters
else if (*pszIndex == _T('%'))
{
*pszReturn++=ConvertHex(pszIndex[1], pszIndex[2]);
pszIndex+=2;
}
// or just copy the character
else
*pszReturn++ = *pszIndex;
pszIndex++;
}
// Terminate the End
*pszReturn = '\0';
return (LPVOID) pszString;
}
DWORD GetValue(LPTSTR szCGI, LPTSTR szName, LPTSTR szValue, DWORD dwValueSize)
{
LPTSTR    szIndex;
LPTSTR    szEnd;
DWORD    dwReturnSize=0;
// Find The Name in the Query String
```

```
szIndex=_tcsstr(szCGI,szName);
// Error: The Name part of the Name value pair doesn't exist
if (!szIndex)
return (0);
// Increase the pointer passed the Name and Get to the Value
szIndex+=_tcslen(szName)+1;
// Find the End of the Value by looking for the '&'
szEnd=_tcschr(szCGI,_T('&'));
// if we find a '&' set it as the end
if (szEnd)
(*szEnd)='\0';
// Remove the CGI Syntax
TranslateCGI(szIndex);
// Calculate the Value Size
dwReturnSize=_tcslen(szIndex);
// Chop the Value if bigger than the Allocation of Value
if (dwReturnSize>dwValueSize)
szIndex[dwValueSize]=_T('\0');
// Assign the Value if there is allocated space
// if no space has been allocated then the caller
// is just looking for the string size
if (szValue)
_tcscpy(szValue,szIndex);
// If we are going to return the size of
// Allocated space we might as well
```

```
// include the Null
return (dwReturnSize+1);
}
// Returns The Length of szValue on successful execution else returns 0
DWORD GetMethod(LPTSTR szName, LPTSTR szValue, DWORD dwValueSize)
{
DWORD    dwBufferSize=0;
DWORD    dwReturnSize=0;
LPTSTR    szQuery=NULL;
// Call GetEnvironmentVariable To get the buffer size
dwBufferSize=GetEnvironmentVariable(_T("QUERY_STRING"),szQuery,dwBufferSize);
// Error: QUERY_STRING doesn't exist
if (!dwBufferSize)
return(0);
// Allocate the Space needed
szQuery = new TCHAR[dwBufferSize];
// Call Again
dwBufferSize=GetEnvironmentVariable(_T("QUERY_STRING"),szQuery,dwBufferSize);
// Get the Value From the Query String
dwReturnSize=GetValue(szQuery,szName,szValue,dwValueSize);
delete szQuery;
return (dwReturnSize);
};
// Returns The Content Length on successful execution else returns 0
DWORD GetContentLength()
```

```
{
DWORD    dwBufferSize=0;
LPTSTR    szContentLength=NULL;
DWORD    dwContentLength;
// Call GetEnvironmentVariable to get the buffer size
dwBufferSize=GetEnvironmentVariable(_T("CONTENT_LENGTH"),
szContentLength,dwBufferSize);
// Error: CONTENT_LENGTH doesn't exist
if (!dwBufferSize)
return(0);
// Allocate the Need Space
szContentLength = new TCHAR[dwBufferSize];
// Call Again
dwBufferSize=GetEnvironmentVariable(_T("CONTENT_LENGTH"),
szContentLength,dwBufferSize);
// Change the String to a usable form
dwContentLength=(DWORD)_ttoi(szContentLength);
delete szContentLength;
return(dwContentLength);
};
// Returns The Length of szValue on successful execution else returns 0
DWORD PostMethod(LPTSTR szName, LPTSTR szValue, DWORD dwValueSize)
{
DWORD    dwBufferSize;
LPTSTR    szContentType=NULL;
```

```
DWORD    dwContentTypeSize=0;
LPTSTR    szPost;
DWORD    dwReturnSize=0;
UINT    nCount;
if (szCache)
{
dwBufferSize=_tcslen(szCache);
// Allocate Some Memory for szPost Plus the NULL
szPost=new TCHAR[dwBufferSize+1];
_tcscpy(szPost,szCache);
}
else
{
// Look at the CONTENT_TYPE to see if it is a POST
// Call GetEnvironmentVariable to get the buffer size
dwContentTypeSize=GetEnvironmentVariable(_T("CONTENT_TYPE"),
szContentType,dwContentTypeSize);
// Error: CONTENT_TYPE doesn't exist
if (!dwContentTypeSize)
return(0);
// Allocate the Need Space
szContentType = new TCHAR[dwContentTypeSize];
// Call Again
GetEnvironmentVariable(_T("CONTENT_TYPE"),szContentType,dwContentTypeSize);
if (!_tcscmp(szContentType,_T("application/x-www-form-urlencoded")))
```

```
{
// Figure out the Size of the String
dwBufferSize=GetContentLength();
if (!dwBufferSize)
return(0);
// Declare the Memory for the String plus the NULL
szPost = new TCHAR[dwBufferSize+1];
nCount=0;
// Read the Standard In
while (!feof(stdin) && (nCount<dwBufferSize))
{
szPost[nCount++]=(TCHAR)_fgetchar();
}
szPost[nCount]=_T('\0');
// Cache the CGI String
szCache=new TCHAR[dwBufferSize+1];
_tcscpy(szCache,szPost);
}
else
{
// Not a POST so return unsuccessfull
return(0);
}
}
// We now have the CGI String, Lets Get the Value
```

```
// Get the Value From the Post String
dwReturnSize=GetValue(szPost,szName,szValue,dwValueSize);
delete szPost;
// If we are going to return the size of
// Allocated space we might as well
// include the Null
return (dwReturnSize);
return(0);
};
// Returns The Length of szValue on successful execution else returns 0
DWORD GetParameter (LPTSTR szName, LPTSTR szValue, DWORD dwValueSize)
{
DWORD    dwBufferSize=0;
LPTSTR    szRequestMethod=NULL;
// Look at the environment variable REQUEST_METHOD
// Call GetEnvironmentVariable to get the buffer size
dwBufferSize=GetEnvironmentVariable(_T("REQUEST_METHOD"),
szRequestMethod,dwBufferSize);
// Error: REQUEST_METHOD doesn't exist
if (!dwBufferSize)
return(0);
// Allocate the Need Space
szRequestMethod = new TCHAR[dwBufferSize];
// Call Again
dwBufferSize=GetEnvironmentVariable(_T("REQUEST_METHOD"),
```

```
szRequestMethod,dwBufferSize);
// It's has to be POST or GET
if (!_tcscmp(szRequestMethod,_T("GET")))
{
delete szRequestMethod;
return(GetMethod(szName,szValue,dwValueSize));
}
if (!_tcscmp(szRequestMethod,_T("POST")))
{
delete szRequestMethod;
return(PostMethod(szName,szValue,dwValueSize));
}
delete szRequestMethod;
return(0);
};
int main( int argc, char *argv[ ], char *envp[ ] )
{
DWORD  dwValueSize=0;
LPTSTR szValue=NULL;
// Header
_tprintf(_T("Status: 200\r\n"));
_tprintf(_T("Content-type: text/html\r\n"));
_tprintf(_T("\r\n"));
// Body
_tprintf(_T("<HTML><BODY>\n"));
```

```
// Find out how big the name parameter is going to be

dwValueSize=GetParameter(_T("Name"),szValue,dwValueSize);

// Allocate enough space for The Value of Name

szValue=new TCHAR[dwValueSize];

// Get The Value Again this time with a big enough buffer

dwValueSize=GetParameter(_T("Name"),szValue,dwValueSize);

// Display the Name

if (dwValueSize)

_tprintf(_T("Name : %s\n"),szValue);

delete szValue;

_tprintf(_T("<BR>\n"));

// Do it all Again for Age

dwValueSize=GetParameter(_T("Age"),szValue,dwValueSize);

szValue=new TCHAR[dwValueSize];

dwValueSize=GetParameter(_T("Age"),szValue,dwValueSize);

if (dwValueSize)

_tprintf(_T("Age : %s\n"),szValue);

_tprintf(_T("</BODY></HTML>\n"));

delete szValue;

// Clean the Cache

if (szCache)

delete     szCache;

return(0);

}
```

The Main function calls GetParameter() twice, once with name and once with age. The return will be the value of name and age as passed in by the form. If no value is present or there is an error, the GetParameter() will return zero;

otherwise, it will return the character length of the value.

With GetParameter(), no matter what method is used in the form, the value of the named variable will be returned. GetParameter() looks at the environmental variable REQUEST_METHOD to figure out if the method is a POST or a GET. If it is a POST method, PostMethod() is called. In PostMethod(), the environment variable CONTENT_TYPE is checked to make sure that the post is coming from a form. CONTENT_LENGTH is also checked so that the right size string can be allocated. The CGI string is then read from standard in as a file would be read. Notice that in PostMethod(), you cache the CGI string returning from standard input; standard input can be read only once. If GetParameter() detects that the GET method is called, then GetMethod() is executed. GetMethod() loads the CGI script from QUERY_STRING. Try experimenting with the FORM method, changing the method in lst13_6.htm to POST. Type lst13_6.htm into your browser address space. You will need to refresh to get the changes. Now, try resubmitting the data.

Both GetMethod() and PostMethod() call GetValue() after the CGI string is acquired. GetValue() separates the value that is associated with the name from the CGI string. After the value is separated, it's passed to TranslateCGI(). TranslateCGI() changes + to spaces and translates hexadecimal characters.

Notice that TCHAR and _t windows functions were used wherever possible. This allows the code to be compiled either in MBCS or UNICODE.

It's important to consider the length of the strings you allocate to hold the data coming in. Remember that real users mean real data. Consider an input tag that sends in first names. On the Web, users from all over the world can access your page, so their first names might be longer than you expect. If the data overrides the memory you have allocated, the CGI script could crash, causing the server to crash. To solve this problem, you can limit the data coming in by using the MAXLENGTH attribute on input tags. You can handle any length value like GetParameter(). Also, make sure that users cannot send in parameters that make your CGI script crash. Because users can type anything in the URL address of their browser, make sure to test all possibilities. For instance, if the standard URL address and CGI string look like this:

```
http://MYMACHINE/scripts/mycgi.exe?Name=John
```

and the a user types this instead

```
http://MYMACHINE/scripts/mycgi.exe?&Name&Name=&&&&&&
```

can your CGI script handle it without crashing?

Debugging C++ CGI Scripts

One of the disadvantages of using C++ CGI scripts is that you have to debug them. The optimal way to test CGI scripts is to put them into the scripts directory and have the server call them. This way, they are called just as they would be in practice. Microsoft Developers Studio has an excellent debugging environment, but in order to use the debugger, Microsoft Developers Studio has to call the executable you're testing. Here lies the dilemma: either to have the server call the CGI scripts without a debugger or have Microsoft Developers Studio call the CGI script without the benefit of the server.

The Web Server Debugging

If the server calls the CGI scripts and there is an ASSERT or an access violation, the process running the CGI script hangs. This usually causes the server not to return a header, leaving the browser waiting until it times out. The reason an ASSERT or an access violation hangs is that it tries to initialize a message box without a valid window handle. The only solution is to kill the process. With the server calling the CGI script, other errors such as returning the wrong output are equally as hard to debug. For instance, if no output is returned by the CGI script, the browser displays the screen shown in Figure 13.1.

Figure 13.1. Bad return.

This type of screen leaves the programmer no information for debugging. Like an ASSERT or an access violation, inserting a DebugBreak() into the CGI script also hangs the process.

The solution is to redirect the output for ASSERT, warnings, or errors to an output file. The StartDebugging() and StopDebugging() procedures in Listing 13.7 create a file called error.log, and all warnings, errors, and ASSERTs will be written to it. StartDebugging() opens the file and redirects the output. StopDebugging() closes the file handle. Add StartDebugging to the beginning of Main() and StopDebugging() to the end of Main().

Listing 13.7. Redirecting debugging information to a file.

```
void StartDebugging()

{

#ifdef _DEBUG

hFile= CreateFile("error.log",GENERIC_WRITE,

FILE_SHARE_WRITE,NULL,OPEN_ALWAYS,

FILE_ATTRIBUTE_NORMAL,NULL);

if (hFile)

{

_CrtSetReportMode(_CRT_WARN, _CRTDBG_MODE_FILE);

_CrtSetReportFile(_CRT_WARN, hFile);

_CrtSetReportMode(_CRT_ERROR, _CRTDBG_MODE_FILE);

_CrtSetReportFile(_CRT_ERROR, hFile);

_CrtSetReportMode(_CRT_ASSERT, _CRTDBG_MODE_FILE);

_CrtSetReportFile(_CRT_ASSERT, hFile);
```

```
}

_RPT0(_CRT_WARN, "Start Debug Reporting\r\n" );

#endif

};

void StopDebugging()

{

#ifdef _DEBUG

_RPT0(_CRT_WARN, "Stop Debug Reporting\r\n" );

if (hFile)

CloseHandle(hFile);

#endif

};

int main( int argc, char *argv[ ], char *envp[ ] )

{

StartDebugging();

// The Code

StopDebugging();

return(0);

}
```

Notice that _RPT0() can be used to write to error.log. This function could be used to mark entrances and exits of procedures, output variables, and other information for debugging. The error file will not be created with release builds and shouldn't be used with multiple processes running.

Tip

Unless you precede the error filename with the full path, the file will be created in the scripts directory. This could be a security issue if your users can read the scripts directory. Either redirect the output out of the Web space or make sure that the directory security is set to execute only.

Microsoft Developer Studio

The other option for debugging your C++ CGI script is to call the CGI script from Microsoft Developer Studio. The advantage of this, compared to having the Web server call the script, is that you can set break points and view variables with Microsoft Developer Studio. Problems start to arise, however, when the input to your CGI script is considered. Because the CGI script retrieves its output from environment variables set by the server, environment variables need to be set to debug. Microsoft Developer Studio doesn't have an easy way to set environment variables, so either the programmer needs to set the variables in the System Properties or the variables need to be set in the program. Setting the variables in the System Properties requires the programmer to open the control panel, change the variables, and restart Microsoft Developer Studio. It's easier to set the variables in the program, recompile the program, and run it from the debugger. Here is an example of setting the variables for Listing 13.8.

Listing 13.8. Inserts for the Setting of the Variables

```
int main( int argc, char *argv[ ], char *envp[ ] )

{

SetEnvironmentVariable("QUERY_STRING","Name=John+Doe&Age=25");

SetEnvironmentVariable("REQUEST_METHOD","GET");

...
```

This is also a great deal of trouble because every time the strings change, you need to recompile.

The POST method poses another problem. Microsoft Developer Studio does not allow you to pipe standard input into a program that you are running on the debugger. Because the Web server sends its information through standard input into a program, there is really no way to test the POST method with Microsoft Developer Studio.

There is no surefire, easy method for testing C++ CGI scripts with either the Web server or Microsoft Developer Studio. These issues get resolved with ISAPI Server Extensions because Microsoft Developer Studio can handle Server Extensions DLLs better.

Summary

The Common Gateway Interface (CGI) is a standardized parameter passing syntax. CGI scripts are programs that are executed by the Web server, that read a CGI parameter string, and that output to the client. Because the data returned to the browser differs for every execution, CGI scripts create dynamic Web pages. A CGI script can be written in any language the Web server can execute. The server passes parameters to the CGI script with environment variables and standard input, and the CGI script passes the output to the server with standard output. A good CGI script can read the CGI sting passed in by the GET and POST method. Although debugging CGI script is not straightforward, you can use them to create dynamic and interactive Web pages quickly.


TOC

Chapter 14

ISAPI Server Applications

by Wayne Berry

HTTP Server Extensions and HTTP Server Filters are Internet Server Applications (ISA). HTTP Server Extensions are CGI scripts on steroids. They execute faster, are easier to debug, and are easier to interface then CGI scripts. One of the reasons for their faster speed is that they remain cached in memory. This means that they do not have to load the Server Extension for every request, as do CGI scripts. Another reason for their speed is that all requests run within the same process; whereas every request of a CGI creates a separate process. Server Extensions are easier to debug because they are DLLs (Dynamic Link Libraries), and Microsoft Developer Studio handles DLL debugging better than the debugging of executables. CGI parameters passed to a Server Extension are packed into a structure, making them easier to access.

How HTTP Server Extensions Work

When a request is made to the server to execute an HTTP Server Extension, the request is passed on to the Server Extension. Before the server passes the request to the extension, it creates a thread, and each request has a separate thread. All HTTP Server Extensions have two common procedures that the HTTP Server calls: One is the version of the HTTP Server Extension; the other is the entrance for the thread. The thread enters the Server Extension by way of the HttpExtensionProc() procedure. The Server retrieves the version number of the HTTP Server Extension from GetExtensionVersion(). Dynamic link libraries built with these two APIs are ISAPI compliant.

Warning

If your server is expected to have a high volume of activity, creating a thread for every request will bog down your machine. In later versions of the IIS, you will be able to make a registry setting to limit the number of threads created. Until that time, it's recommended that you create a thread pool within the Server Extension to keep your machine from being thread bound.

Because there is a thread for each process, making threads execute quickly and return quickly is a must. Users do not want to wait on other users for their threads to get processed. When writing the Server Extension code, make sure that the threads are not waiting for other threads.

You also need to make sure that the code you write is thread safe. Static data that is shared between threads needs to be protected. Also, make sure that DLLs you're calling are thread safe. Currently, ODBC (Open Database Connectivity) is thread safe, but DAO (Data Access Object) is not.

Creating an HTTP Server Extension with Microsoft Developer Studio

Microsoft Developer Studio allows you to create HTTP Server Extensions by using a wizard. Follow these steps:

1. 1. From the Menu bar, click File|New.
2. 2. Select Project Workspace from the New List and select OK (see Figure 14.1).

Figure 14.1. Creating a ISAPI server extension using the New Project wizard.

1. 4. From the Type list, go to the bottom and select ISAPI Extension Wizard.
2. 5. In the Location edit box, change the location to the root of your scripts directory. The default location of the IIS is c:\inetsrv\scripts. This allows you to debug the Server Extension in the same directory that you compile it.
3. 6. Type in the name of the first Listing, **lst14_01**, in the Name edit box. The wizard creates the entire listing for you.
4. 7. Click Create, and you will see the dialog box in Figure 14.2.

Figure 14.2. The ISAPI Extension Wizard Dialog Box

1. 8. Leave the defaults and click Finish.

Tip

Leaving MFC as a shared DLL will increase the loading speed of your DLL. It will also help shared resources if more then one HTTP Server Extension is being run at the same time. But you will need to remember to copy mfc40.dll and msvcrt40.dll to your production server when you copy your release build of HTTP Server Extensions.

---

1. 9. Click OK to create the files.

Listing 14.1 shows what the Server Extension wizard created for the source file (lst14_1.cpp).

Listing 14.1. lst14_1.cpp.

```
// LST14_01.CPP - Implementation file for your Internet Server
//    lst14_01 Extension
#include <afx.h>
#include <afxwin.h>
#include <afxisapi.h>
#include "resource.h"
#include "lst14_01.h"
////////////////////////////////////////////////////////////////////////
// command-parsing map
BEGIN_PARSE_MAP(CLst14_01Extension, CHttpServer)
// TODO: insert your ON_PARSE_COMMAND() and
// ON_PARSE_COMMAND_PARAMS() here to hook up your commands.
// For example:
ON_PARSE_COMMAND(Default, CLst14_01Extension, ITS_EMPTY)
ON_PARSE_COMMAND(Hello, CLst14_01Extension, ITS_EMPTY)
```

```
ON_PARSE_COMMAND(GetName, CLst14_01Extension, ITS_PSTR)
ON_PARSE_COMMAND_PARAMS("Name")
DEFAULT_PARSE_COMMAND(Default, CLst14_01Extension)
END_PARSE_MAP(CLst14_01Extension)
////////////////////////////////////////////////////////////////////
// The one and only CLst14_01Extension object
CLst14_01Extension theExtension;
////////////////////////////////////////////////////////////////////
// CLst14_01Extension implementation
CLst14_01Extension::CLst14_01Extension()
{
TRACE("Constructor\n");
m_pnCounter = new UINT;
(*m_pnCounter) = 0;
}
CLst14_01Extension::~CLst14_01Extension()
{
TRACE("Destructor\n");
delete m_pnCounter;
}
BOOL CLst14_01Extension::GetExtensionVersion(HSE_VERSION_INFO* pVer)
{
// Call default implementation for initialization
CHttpServer::GetExtensionVersion(pVer);
// Load description string
TCHAR sz[HSE_MAX_EXT_DLL_NAME_LEN+1];
ISAPIVERIFY(::LoadString(AfxGetResourceHandle(),
```

```
IDS_SERVER, sz, HSE_MAX_EXT_DLL_NAME_LEN));
_tcscpy(pVer->lpszExtensionDesc, sz);
return TRUE;
}
////////////////////////////////////////////////////////////////////
// CLst14_01Extension command handlers
void CLst14_01Extension::Default(CHttpServerContext* pCtxt)
{
StartContent(pCtxt);
WriteTitle(pCtxt);
*pCtxt << _T("This default message was produced by the Internet");
*pCtxt << _T(" Server DLL Wizard. Edit your CLst14_01Extension::Default()");
*pCtxt << _T(" implementation to change it.\r\n");
EndContent(pCtxt);
}
void CLst14_01Extension::Hello(CHttpServerContext* pCtxt)
{
StartContent(pCtxt);
WriteTitle(pCtxt);
*pCtxt << _T("Hello World\r\n");
EndContent(pCtxt);
}
void CLst14_01Extension::GetName(CHttpServerContext* pCtxt, LPCTSTR pName)
{
StartContent(pCtxt);
WriteTitle(pCtxt);
*pCtxt << _T("Name: ");
```

```
*pCtxt << pName;
*pCtxt << _T("<BR>\r\n");
EndContent(pCtxt);
}
////////////////////////////////////////////////////////////////////
// If your extension will not use MFC, you'll need this code to make
// sure the extension objects can find the resource handle for the
// module.  If you convert your extension to not be dependent on MFC,
// remove the comments arounn the following AfxGetResourceHandle()
// and DllMain() functions, as well as the g_hInstance global.
/****
static HINSTANCE g_hInstance;
HINSTANCE AFXISAPI AfxGetResourceHandle()
{
return g_hInstance;
}
BOOL WINAPI DllMain(HINSTANCE hInst, ULONG ulReason,
LPVOID lpReserved)
{
if (ulReason == DLL_PROCESS_ATTACH)
{
g_hInstance = hInst;
}
return TRUE;
}
****/
```

Here is the header file (lst14_01.h) that got created:

```
// LST14_01.CPP - Implementation file for your Internet Server

//    lst14_01 Extension

class CLst14_01Extension : public CHttpServer

{

public:

UINT *m_pnCounter;

CLst14_01Extension();

~CLst14_01Extension();

BOOL GetExtensionVersion(HSE_VERSION_INFO* pVer);

// TODO: Add handlers for your commands here.

// For example:

void Default(CHttpServerContext* pCtxt);

void Hello(CHttpServerContext* pCtxt);

void GetName(CHttpServerContext* pCtxt, LPCTSTR pName);

DECLARE_PARSE_MAP()

};
```

Now compile the debug version of the newly created HTTP Server Extension.

Debugging the HTTP Server Extension

The best way to debug the Server Extension is to run the server through Microsoft Developer Studio. To do this, you need to be on the same machine that the IIS is currently running on. Make sure that this server is being used only by you; no other Web pages should be served from it. It is advisable to run another copy of the Microsoft Developer Studio to make changes to and recompile your Server Extension. Use your browser on the same machine to call the Server Extension.

Correct Permissions

It's important to remember that the operating system believes that there are two users in the debugging scenarios. The first user, you, is debugging the IIS with the Microsoft Developer Studio. The second user, you, is coming in from the network using a browser and calling the ISAPI Server Extension. The first user has your login name and has to be an administrator. The second user is the IUSR_*MACHINENAME* where *MACHINENAME* is the name of your machine. The second user gets set up by the IIS when it installs. This name can be changed in the Internet service manager; IUSR_*MACHINENAME* is the

default.

In the debugging scenario, the second user permissions are adequate. However, special permission is needed for the first user to run the IIS under Microsoft Developer Studio. If you are not in the Administrator group on this machine, find an administrator and have that administrator make you one. After you are an administrator, you need to give yourself a few more permissions. Follow these steps:

1. 1. Make sure that you are logged on with administrator permissions.
2. 2. Choose Administrative Tools|User Manager for Domains.
3. 3. Make sure that User Manager is administrating the local machine and not the domain.
4. 4. Select Policies|User Rights.
5. 5. Click the Show Advanced User Rights checkbox in the bottom-right corner.
6. 6. From the Grant Select Box on the right hand side, choose "Act as part of the operating system."
7. 7. Click Add and add yourself.
8. 8. Click OK.
9. 9. From the Grant Select Box choose "Generate security audits."
10. 10. Click Add and add yourself.
11. 13. Click OK to exit Add Users and Groups Dialog.
12. 14. Click OK to exit the User Rights Policy Dialog.
13. 15. Exit User Manager.
14. 16. Log off and log back on.

Disabling the IIS

In order to run the IIS on Microsoft Developer Studio, you will need to stop running the default IIS. You can do this by going into the Internet Service Manager and stopping the server. You might also want to disable the server from restarting if you reboot the machine. To do this, you will need to do the following:

1. 1. Go into Control Panel.
2. 2. Choose Services.
3. 3. Choose World Wide Web Publishing Service from the Services List.
4. 4. Click Startup.
5. 5. In the Startup dialog box, choose Manual and click OK.
6. 6. Click Close and exit the Control Panel.

Disabling Caching

As mentioned before, the IIS Server will cache the Server Extension upon the first request and keep it loaded in memory. It's important to disable caching when debugging the Server Extension. With the Server Extension loading and unloading for each request, Microsoft Developer Studio is able to detect some types of memory leaks.

Here is the way to modify the registry so that the IIS will not cache the Server Extensions.

1. 1. Open the Registry editor, by executing regedit.exe.
2. 2. Choose HKEY_LOCAL_MACHINE of the registry tree and expand the branch.
3. 3. Then expand the branches leading to this registry key at the location, SYSTEM\CurrentControlSet\Services\W3SVC\Parameters.

4. 4. There is a value within Parameters called CacheExtensions. Set this to 0.

Running the IIS in Microsoft Developer Studio

Now that you have adequate permissions and have disabled the default IIS and extension caching, you can run the IIS in Microsoft Developer Studio by following these steps:

1. 1. Start another copy of Microsoft Developer Studio besides the one open for compiling the Server Extension.
2. 2. From the menu bar, choose File|Open Workspace.
3. 3. From the Open Workspace dialog's File of Type list, select Executable files (*.exe).
4. 4. Open the IIS executable c:\inetsrv\server\inetinfo.exe.
5. 5. From the menu bar of Microsoft Developer Studio, click Build|Settings.
6. 6. Click the Debug tab.
7. 7. Select General from the Category drop-down box.
8. 8. In the Program argument's edit box, type **-e W3Svc**.
9. 9. Select "Additional DLLs" from the Category drop-down box.
10. 10. In the Modules box, choose the debug HTTP Server lst14_01.dll. It should be at c:\inetsrv\lst14_01\debug\lst14_01.dll.
11. 11. Click OK and exit the Settings dialog.
12. 12. From the Menu bar, choose File|Save All.
13. 13. When asked Do you want to save project workspace information for C:\inetsrv\server\inetinfo.mdp, choose Yes. This way, whenever you want to run the IIS Server Extension in the Microsoft Developer Studio, you can just open C:\inetsrv\server\inetinfo.mdp instead of performing all the previous steps.
14. 14. From the Menu bar, choose Build.
15. 15. Click on Execute inetinfo.exe.
16. 16. Every time you execute inetinfo.exe, you will be asked 'C:\inetsrv\server\inetinfo.exe' does not contain debugging information. Do you want to continue? Choose Yes.

Now IIS is running in the debugger of IIS. You can debug the server extension like any other DLL.

Running the Example Created by Extension Wizard

With the IIS running in the debugger, open your browser and type in the address of the Server Extension DLL. The example you just created should be at the URL

`http://MYMACHINE/scripts/lst14_01/debug/lst14_01.dll`

where *MYMACHINE* is the name of your machine. The page returned from the Server Extension should look like Figure 14.3.

Figure 14.3. Listing 14.1 Extension DLL displayed in the browser.

The Default Memory Leak

In the default set up by the Extension wizard, there is a bug that I want to expose right away.

After executing the Server Extension, look at the output window of the debugger running the IIS. Notice that there is a line that reads

```
Detected Memory leaks!
```

The reason for the memory leak is that the destructor for CHttpServer class is not being called.

This is a bug in the Microsoft Foundation Classes shipping with Microsoft Visual C++ 4.1 that will be fixed in Microsoft Visual C++ 4.2. To fix the bug in Microsoft Foundation Classes that ship with Microsoft Visual C++ 4.1, you must add this line in lst14_01.cpp:

```
CWinApp BugFix;
```

It needs to go right after the static declaration of the extension class

```
CLst14_01Extension theExtension;
```

This creates a CWinApp object, which you never use. The problem is fixed because the compiler links with a different library when a CWinApp object is present. The different library doesn't contain the bug.

Recompile the Server Extension. Every time you recompile the server extension, you will need to stop debugging the IIS. This is because the Server Extension symbols file is loaded into the Microsoft Developer Studio while the IIS is debugging. This is the reason that I recommend two open Microsoft Developer Studios. With two Microsoft Developer Studios, it's easier to switch between compiling and debugging.

Execute the IIS again under Microsoft Developer Studio. Notice that the memory leak is gone.

Two Ways of Calling Server Extensions

You can call the Server Extension in two ways. One way is to call the server extension by using the full path name as described in the preceding section. The only problem with this is that it isn't very clean. The user knows that you are running an IIS using an ISA. The second way involves assigning an extension to the ISA in the registry. The request to the server would use this extension to signify that it wants to run the Server Extension. Microsoft uses this technique with its Internet Database Connector. Microsoft in IIS assigns the extension .idc to a DLL named httpodbc.dll. Whenever a request is made with an extension of .idc, the Internet Database Connector (httpodbc.dll) is called. The Internet Database Connector is a good example of when you would want to assign extensions to ISA. The request to the Internet Database Connector looks like a request to a file; for example:

```
http://MYMACHINE/scripts/query.idc
```

The server doesn't load query.idc; instead, it sees the .idc extension and passes the request on to the Internet Database

Connector. The Internet Database Connector in turn loads the file called query.idc and processes it in a special way. Here is how you assign an unique extension to your ISAPI Server Extension:

1. 1. Open the Registry editor.
2. 2. Choose HKEY_LOCAL_MACHINE.
3. 3. Then choose the registry key Script Map at this location SYSTEM\CurrentControlSet\Services\W3SVC\Parameters\Script Map.
4. 4. Create a new string value named after your extension and with a value that points to the path of your Server Extension DLL. Use the Internet Database Connector as an example.
5. 5. Reboot the Machine.

Note

After you create a mapping to the extension, that extension will be called whether or not there is a file by that name. The DLL can handle each request for a file, with the unique extension, like a separate file, without actually loading the file from the disk. This creates the impression that the user is skipping from one file to another when there really aren't any files at all. For example if you have a server extension and you map it the file extension .shp, then the server extension would be called no matter if you requested page1.shp or page2.shp. The file page1.shp does not need to exist on your hard drive for the server extension to be called.

Using the Parse Map

The *parse map* is a feature of Microsoft's CHttpServer class. It is used to bind specific procedures with requests to the Server Extension. Each request to the Server Extension must have a name that is the name of the procedure in the Server Extension it's calling. For example, if you have an ISAPI Server Extension that is using parse maps and that server extension has two procedures called Page1 and Page2, you will have to reference in the URL to call them. The references might look like this:

```
http://MYMACHINE/scripts/MYDLL.dll?Page1

http://MYMACHINE/scripts/MYDLL.dll?Page2
```

MYMACHINE is the name of your machine and *MYDLL* is the ISAPI Server Extension.

Besides calling the name of the procedure in the request, the client must also supply a specific matching parameter set in the right order. Because of these requirements, the parse map approach is difficult to use.

The Default Procedure

The Extension wizard writes a default procedure into the parse map. This procedure is called Default and takes no parameters. When the Extension DLL binds all requests that do not specify a procedure to the Default procedure. The entry in the parse map looks like this:

```
ON_PARSE_COMMAND(Default, CLst14_01Extension, ITS_EMPTY)
```

Notice the second command of the ON_PARSE_COMMAND is ITS_EMPTY. This is used to signify that the Default command takes no parameters.

The Default procedure looks like this:

```
void CLst14_01Extension::Default(CHttpServerContext* pCtxt)

{

StartContent(pCtxt);

WriteTitle(pCtxt);

*pCtxt << _T("This default message was produced by the Internet");

*pCtxt << _T(" Server DLL Wizard. Edit your CLst14_01Extension::Default()");

*pCtxt << _T(" implementation to change it.\r\n");

EndContent(pCtxt);

}
```

Notice that a CHttpServerContext class is passed into the Default procedure. All procedures that are called from the parse map must accept the CHttpServerContext class. By using the << operator of the CHttpServerContext class, you can pipe information to the client. Notice that before any information is written to the CHttpServerContext class, the CHttpServer StartContent() method is called. All this method does is write the <HTML> and <BODY> tags to the client. The EndContent() method writes the end <BODY> and end <HTML> tags to the client. These methods are frivolous because they can be replaced with

```
*pCtxt << _T("<HTML><BODY>");
```

and

```
*pCtxt << _T("</HTML></BODY>");
```

To request the Default procedure, type the server path name of the DLL and the method to call into the browser like this:

```
http://MYMACHINE/scripts/lst14_01/debug/lst14_01.dll?Default
```

In this line, *MYMACHINE* is the name of your machine.

A second way to call the Default procedure is by using the following:

```
http://MYMACHINE/scripts/lst14_01/debug/lst14_01.dll
```

This works because the Default procedure has also been defined in the parse map as the procedure to call if no procedure is

named. The parse map entry to do this is

```
DEFAULT_PARSE_COMMAND(Default, CLst14_01Extension)
```

Creating a Second Method

Let's create another method called Hello. First make the entry in the parse map like this:

```
ON_PARSE_COMMAND(Hello, CLst14_01Extension, ITS_EMPTY)
```

Then create a method called Hello that returns nothing and takes a pointer to CHttpServerContext:

```
void CLst14_01Extension::Hello(CHttpServerContext* pCtxt)
{
StartContent(pCtxt);
WriteTitle(pCtxt);
*pCtxt << _T("Hello World!\r\n");
EndContent(pCtxt);
}
```

Make sure that you add the method to the CLst14_01Extension class definition, also.

To call the new method, type the server path, the name of the DLL, and the method that you wish to call, into the browser address space, like this:

```
http://MYMACHINE/scripts/lst14_01/debug/lst14_01.dll?Hello
```

where *MYMACHINE* is the name of your machine.

Your screen should look like Figure 14.4.

Figure 14.4. Browser view of the Hello World Method

Getting the CGI Parameters

Create another method called GetName(). Besides a pointer to CHttpServerContext, GetName() also takes a character string. GetName() should look like this:

```
void CLst14_01Extension::GetName(CHttpServerContext* pCtxt, LPCTSTR pName)

{

StartContent(pCtxt);

WriteTitle(pCtxt);

*pCtxt << _T("Name: ");

*pCtxt << pName;

*pCtxt << _T("<BR>\r\n");

EndContent(pCtxt);

}
```

The entry in the parse map should look like this:

```
ON_PARSE_COMMAND(GetName, CLst14_01Extension, ITS_PSTR)
```

Make sure that you add the method to the CLst14_01Extension class definition also.

To call the new method, type the server path, the name of the DLL, the method to call, and the string you want to enter into the browser address space, like this:

```
http://MYMACHINE/scripts/lst14_01/debug/lst14_01.dll?GetName?Joe
```

Here, *MYMACHINE* is the name of your machine.

The output from the browser should look like Figure 14.5.

Figure 14.5. The output from the browser.

Using Forms with Parse Maps

As discussed in the preceding chapter, in writing CGI scripts, you need to include code to handle both the GET and POST methods for HTML forms. By using a Server Extension, underlying MFC code handles the differences between these methods, and the programmer doesn't need to do anything. To demonstrate this, let's create an HTML page that calls the Server Extension with the GET method.

Listing 14.2. HTML to Request Listing 14.1.

```
<HTML>
```

```
<BODY>

<FORM ACTION="http://dartbldn/scripts/lst14_01/debug/lst14_01.dll?GetName"

[ic:ccc]METHOD=POST>

<INPUT TYPE=TEXT NAME="Name" VALUE="">

<INPUT TYPE=SUBMIT>

</FORM>

</BODY>

</HTML>
```

Tip

Parse maps work best with forms that send POST methods. Some browsers truncate the ?GetName with a GET method, causing the Server Extension to fail.

Default Parameters

Try calling the GetName() method without a name, like this:

```
http://MYMACHINE/scripts/lst14_01/debug/lst14_01.dll?GetName
```

This should cause an error because you don't tell the parse map what to use for default parameters.

Try adding the following line right after the ON_PARSE_COMMAND that binds the GetName() method:

```
ON_PARSE_COMMAND_PARAMS("string=N/A")
```

Note

It's important to note here that the ON_PARSE_COMMAND_PARAMS line does not contain a reference to a method. The compiler knows which ON_PARSE_COMMAND_PARAMS line is associated to which method by looking at the ON_PARSE_COMMAND line proceeding the ON_PARSE_COMMAND_PARAMS line in the parse map. The ON_PARSE_COMMAND line then contains the needed reference to the method.

Recompile and try the preceding URL again. This time the browser should look like Figure 14.6.

Figure 14.6. The request to GetName through the Browser

The Problems with Parse Maps

Parse maps are good for simple things, but they are not suited for complex Server Extensions. One problem is that from a programmer's perspective, there are many entrances to the DLL. If the programmer wants to add specific code that needs to be called whenever a thread enters a DLL, it must be added to each method that is bound to the parse map. This makes things such as a thread pool harder to program. A second problem is that the parameters must be sent in a particular order. This works well for the disciplined programmer but is not the way CGI applications are accustomed to receiving their CGI parameters. Besides sending parameters in a particular order (that matches the parse map), you must also send the right number of parameters. This limitation makes it more difficult to implement certain types of Server Extensions.

Beyond Parse Maps

You can program Server Extensions like you program CGI scripts, with one entrance for every thread. This technique still leaves you with the performance of Server Extensions without the headaches of parse maps. You need to remove the parse maps and write a little code to grab the CHttpServerContext for yourself.

One Entrance Server Extension

Create a Server Extension with the Extension Wizard call it lst14_03.dll.

The created code for the header file (lst14_03.h) will look like this :

Listing 14.2. lst14_03.cpp.

```
// LST14_03.CPP - Implementation file for your Internet Server

//    lst14_03 Extension

class CLst14_03Extension : public CHttpServer

{

public:

UINT *m_pnCounter;

CLst14_03Extension();

~CLst14_03Extension();

BOOL GetExtensionVersion(HSE_VERSION_INFO* pVer);

// TODO: Add handlers for your commands here.

// For example:

void Default(CHttpServerContext* pCtxt);
```

```
DECLARE_PARSE_MAP()

};
```

The created code for the header file (lst14_03.h) will look like this :

```
// LST14_03.CPP - Implementation file for your Internet Server
//    lst14_03 Extension

#include <afx.h>
#include <afxwin.h>
#include <afxisapi.h>
#include "resource.h"
#include "lst14_03.h"

////////////////////////////////////////////////////////////////////
// command-parsing map
BEGIN_PARSE_MAP(CLst14_03Extension, CHttpServer)
// TODO: insert your ON_PARSE_COMMAND() and
// ON_PARSE_COMMAND_PARAMS() here to hook up your commands.
// For example:
ON_PARSE_COMMAND(Default, CLst14_03Extension, ITS_EMPTY)
DEFAULT_PARSE_COMMAND(Default, CLst14_03Extension)
END_PARSE_MAP(CLst14_03Extension)

////////////////////////////////////////////////////////////////////
// The one and only CLst14_03Extension object
CLst14_03Extension theExtension;

////////////////////////////////////////////////////////////////////
// CLst14_03Extension implementation
CLst14_03Extension::CLst14_03Extension()
{
}
```

```
CLst14_03Extension::~CLst14_03Extension()
{
}
BOOL CLst14_03Extension::GetExtensionVersion(HSE_VERSION_INFO* pVer)
{
// Call default implementation for initialization
CHttpServer::GetExtensionVersion(pVer);
// Load description string
TCHAR sz[HSE_MAX_EXT_DLL_NAME_LEN+1];
ISAPIVERIFY(::LoadString(AfxGetResourceHandle(),
IDS_SERVER, sz, HSE_MAX_EXT_DLL_NAME_LEN));
_tcscpy(pVer->lpszExtensionDesc, sz);
return TRUE;
}
////////////////////////////////////////////////////////////////////
// CLst14_03Extension command handlers
void CLst14_03Extension::Default(CHttpServerContext* pCtxt)
{
StartContent(pCtxt);
WriteTitle(pCtxt);
*pCtxt << _T("This default message was produced by the Internet");
*pCtxt << _T(" Server DLL Wizard. Edit your CLst14_03Extension::Default()");
*pCtxt << _T(" implementation to change it.\r\n");
EndContent(pCtxt);
}
////////////////////////////////////////////////////////////////////
// If your extension will not use MFC, you'll need this code to make
```

```
// sure the extension objects can find the resource handle for the
// module.  If you convert your extension to not be dependent on MFC,
// remove the comments arounn the following AfxGetResourceHandle()
// and DllMain() functions, as well as the g_hInstance global.
/****
static HINSTANCE g_hInstance;
HINSTANCE AFXISAPI AfxGetResourceHandle()
{
return g_hInstance;
}
BOOL WINAPI DllMain(HINSTANCE hInst, ULONG ulReason,
LPVOID lpReserved)
{
if (ulReason == DLL_PROCESS_ATTACH)
{
g_hInstance = hInst;
}
return TRUE;
}
****/
```

Now remove the parse map code by removing the following lines from lst14_03.cpp:

```
////////////////////////////////////////////////////////////////////
// command-parsing map
BEGIN_PARSE_MAP(CLst14_03Extension, CHttpServer)
// TODO: insert your ON_PARSE_COMMAND() and
// ON_PARSE_COMMAND_PARAMS() here to hook up your commands.
```

```
// For example:

ON_PARSE_COMMAND(Default, CLst14_03Extension, ITS_EMPTY)

DEFAULT_PARSE_COMMAND(Default, CLst14_03Extension)

END_PARSE_MAP(CLst14_03Extension)
```

Remove the declaration of the parse map from lst14_03.h:

```
DECLARE_PARSE_MAP()
```

Then remove the Default method from lst14_03.cpp:

```
////////////////////////////////////////////////////////////////////

// CLst14_03Extension command handlers

void CLst14_03Extension::Default(CHttpServerContext* pCtxt)

{

StartContent(pCtxt);

WriteTitle(pCtxt);

*pCtxt << _T("This default message was produced by the Internet");

*pCtxt << _T(" Server DLL Wizard. Edit your CLst14_03Extension::Default()");

*pCtxt << _T(" implementation to change it.\r\n");

EndContent(pCtxt);

}
```

Remove the definition of the default from lst14_03.h:

```
// TODO: Add handlers for your commands here.

// For example:

void Default(CHttpServerContext* pCtxt);
```

Now that you've removed the parse maps, you need to add the code that fixes the default memory leak. Add this line to lst14_03.cpp:

```
CWinApp BugFix;
```

It needs to go right after the static declaration of the extension class

```
CLst14_01Extension theExtension;
```

To create the one entrance into the Server Extension, you need to overload the method CallFunction() in your CHttpServer class. Add this code to lst14_03.cpp:

```
int CLst14_03Extension::CallFunction(CHttpServerContext*

[ic:ccc]pCtxt,LPTSTR pszQuery, LPTSTR pszCommand)

{

int nRet=callOK;

ISAPIASSERT(pCtxt->m_pStream == NULL);

pCtxt->m_pStream = ConstructStream();

if (pCtxt->m_pStream == NULL)

nRet = callNoStream;

else

{

pCtxt->m_pStream->InitStream();

Entrance(pCtxt,pszQuery);

}

return(nRet);

}
```

You also need to declare the method in the definition of the class. Add this code to lst14_03.h:

```
int CallFunction(CHttpServerContext* pCtxt,LPTSTR pszQuery, LPTSTR pszCommand);
```

Notice that a new method called Entrance is called in CallFunction(). You need to add the Entrance method, also. For now, let's make it return Hello World!:

```
void CLst14_03Extension::Entrance(CHttpServerContext* pCtxt,LPTSTR pszQuery)

{
```

```
(*pCtxt) << _T("<HTML><BODY>Hello World!</BODY</HTML>");

};
```

You also need to declare the method in the definition of the class. Add this code to lst14_03.h:

```
void Entrance(CHttpServerContext* pCtxt,LPTSTR pszQuery);
```

This is the minimum amount of code needed to make a single entrance Server Extension. All other examples in this chapter will be based on this starting code.

Here is what you should have after making all the changes :

Listing 14.3. lst14_#1.h.

```
Header (lst14_01.h):

// LST14_03.CPP - Implementation file for your Internet Server

//    lst14_03 Extension

class CLst14_03Extension : public CHttpServer

{

public:

UINT *m_pnCounter;

CLst14_03Extension();

~CLst14_03Extension();

BOOL GetExtensionVersion(HSE_VERSION_INFO* pVer);

int CallFunction(CHttpServerContext* pCtxt,LPTSTR pszQuery, LPTSTR pszCommand);

void Entrance(CHttpServerContext* pCtxt,LPTSTR pszQuery);

};
```

Source:

```
// LST14_03.CPP - Implementation file for your Internet Server

//    lst14_03 Extension

#include <afx.h>

#include <afxwin.h>
```

```
#include <afxisapi.h>

#include "resource.h"

#include "lst14_03.h"

//////////////////////////////////////////////////////////////////////

// The one and only CLst14_03Extension object

CLst14_03Extension theExtension;

CWinApp BugFix;

//////////////////////////////////////////////////////////////////////

// CLst14_03Extension implementation

CLst14_03Extension::CLst14_03Extension()

{

}

CLst14_03Extension::~CLst14_03Extension()

{

}

BOOL CLst14_03Extension::GetExtensionVersion(HSE_VERSION_INFO* pVer)

{

// Call default implementation for initialization

CHttpServer::GetExtensionVersion(pVer);

// Load description string

TCHAR sz[HSE_MAX_EXT_DLL_NAME_LEN+1];

ISAPIVERIFY(::LoadString(AfxGetResourceHandle(),

IDS_SERVER, sz, HSE_MAX_EXT_DLL_NAME_LEN));

_tcscpy(pVer->lpszExtensionDesc, sz);

return TRUE;

}

int CLst14_03Extension::CallFunction(CHttpServerContext *pCtxt,LPTSTR pszQuery,
LPTSTR pszCommand)
```

```
{

int nRet=callOK;

ISAPIASSERT(pCtxt->m_pStream == NULL);

pCtxt->m_pStream = ConstructStream();

if (pCtxt->m_pStream == NULL)

nRet = callNoStream;

else

{

pCtxt->m_pStream->InitStream();

Entrance(pCtxt,pszQuery);

}

return(nRet);

}

void CLst14_03Extension::Entrance(CHttpServerContext* pCtxt,LPTSTR pszQuery)

{

(*pCtxt) << _T("<HTML><BODY>Hello World!</BODY</HTML>");

};
```

Compile lst14_03.dll. Before you run the IIS in the debugger, you will need to add this new DLL to the IIS project. You can do this by including it in Settings|Additional DLLs, just as you did for lst14_01.dll. Now, run the IIS in the Debugger and execute the Server Extension through your browser.

Why Not Overload HttpExtensionProc?

HttpExtensionProc is the procedure that IIS calls when entering the DLL. It seems logical that if you want to get down to the nitty gritty of Server Extension programming, you would want to overload HttpExtensionProc. This, however, is not the goal; the goal is to have a single entry point into the DLL. HttpExtensionProc calls CallFunction(), but between the two functions, is code to initialize streams and handle some types of errors. Because CallFunction() has both the CHttpServerContext and the CGI parameters, it's easier to overload and use the HttpExtensionProc. CallFunction() also meets the goal because it handles all requests. The parent method of CallFunction() figures out which method to call using the parse map. Overloading CallFunction() removes the parse maps from the Server Extension.

Creating a Thread Pool

Single-entrance server extensions allow the programmer to easily create thread pools. Because a thread is created for every request to the DLL, a Web site that is busy might create enough requests to bog down the machine with threads. This could cause thread lock, making for slow request, or it could cause the operating system to crash. To prevent this type of problem, it's best to use a thread pool within the Server Extension. The type of thread pool that I recommend is one that makes threads wait until other threads are processed, instead of telling the user that the site is too busy to handle the requests.

To create a thread pool, you need to create some protected member variables that will keep track of the number of threads in the DLL.

Create a protected member variable in CLst14_03Extension called m_pnCounter with this line of code:

```
UINT *m_pnCounter;
```

You also need a Critical Section to prevent two threads from accessing the counter variable at the same time. A Critical Section is a section of code in Windows that is restricted to one process. Let's create another protected member variable to handle this:

```
LPCRITICAL_SECTION m_lpCriticalExtension;
```

Initialize the counter in the server extension constructor and delete it in the destructor. Also initialize and delete the Critical Section:

```
CLst14_03Extension::CLst14_03Extension()

{

m_pnCounter = new UINT;

(*m_pnCounter) = 0;

m_lpCriticalExtension = new CRITICAL_SECTION;

InitializeCriticalSection(m_lpCriticalExtension);

}

CLst14_03Extension::~CLst14_03Extension()

{

delete m_pnCounter;

DeleteCriticalSection(m_lpCriticalExtension);

delete m_lpCriticalExtension;
```

```
}
```

Create two new methods, as shown in the following code segment. One method will check to see if the maximum number of threads has been reached; call this method ThreadPool(). Another method will decrement the counter when the thread exits the DLL; call this method LeavePool().

```
void CLst14_03Extension::ThreadPool()

{

BOOL bGoodToGo=FALSE;

while (!bGoodToGo)

{

EnterCriticalSection(m_lpCriticalExtension);

bGoodToGo=((*m_pnCounter)<15);

if(bGoodToGo)

(*m_pnCounter)++;

LeaveCriticalSection(m_lpCriticalExtension);

}

}

void CLst14_03Extension::LeavePool()

{

EnterCriticalSection(m_lpCriticalExtension);

(*m_pnCounter)--;

LeaveCriticalSection(m_lpCriticalExtension);

};
```

Make sure to define the methods in the declaration of the class. The methods also need to be added to Entrance(): one method at the beginning of Entrance, and one at the end, as in the following:

```
void  CLst14_03Extension::Entrance(CHttpServerContext* pCtxt,LPTSTR pszQuery)

{

ThreadPool();

(*pCtxt) << _T("<HTML><BODY>Hello World!</BODY</HTML>");
```

```
LeavePool();

};
```

Considerations When Dealing with Thread Pools

You might have some trouble with this code because it holds the number of threads down to 15. If a lot of requests come simultaneously to your Web space, some of them might time out before being allowed to enter the DLL and get information. This really isn't a problem with the thread pooling; the thread pooling just tries to prevent the operating system from crashing. If users can't get into the DLL in a timely manner, you need to consider either making the DLL return the pages faster by doing performance enhancements, or adding hardware to meet the demand. The number 15 is an arbitrary number. This number should be increased or decreased based on the hardware you are running. Machines with more processors may want to have more threads running at any given time. You also need to take into consideration other Server Extensions' threads.

Notice that the threads are not processed on a first come, first served basis. This is because the code assumes that all processes will be handled in a timely manner and the count is above fifteen, just momentarily. Think of your DLL as a store that is not used to having long lines—if you do have long lines, you need to hire another cashier (add more hardware).

You might want to implement a feature that returns a page saying "Please return later our server is busy". This message would only come up if threads come in while there are more threads than you can handle in the DLL. This makes it look as though you are in control of the situation instead of your server timing out. This works well if you're sure of the number of threads that your server can handle. If you set the maximum thread count too low, you could be turning away people you don't have to.

CGI Parameters and Single Entrance Server Extension

The CGI parameters passed to the server from the browser with both the POST and GET methods are available in the Entrance() method. The variable pszQuery, which is passed into the Entrance() method, contains the CGI parameters. Unlike the parse map, the CGI parameters in pszQuery are still encoded. To read the CGI parameters, you need to add a new method to the class. The two parameters of FindNameValue() are the pszQuery string, plus the name of the 'name=value' pair. FindNameValue() returns the value of the 'name=value'. The memory allocated for the value needs to be deleted by the caller. Here is the method:

```
// The Caller must delete the memory

LPTSTR CLst14_03Extension::FindNameValue(LPCTSTR lpszString,

[ic:ccc]LPCTSTR lpszName,TCHAR cSeparator,TCHAR cDelimiter)

{

LPTSTR lpszIndex;

LPTSTR lpszEnd;

DWORD dwValueSize=0;

LPTSTR lpszValue;
```

```
LPTSTR lpszStringCopy;
LPTSTR lpszNameCopy;
DWORD dwNameLength;
lpszStringCopy = new TCHAR[_tcslen(lpszString)+1];
_tcscpy(lpszStringCopy,lpszString);
lpszNameCopy = new TCHAR[_tcslen(lpszName)+2];
_tcscpy(lpszNameCopy,lpszName);
dwNameLength=_tcslen(lpszNameCopy);
lpszNameCopy[dwNameLength]=cSeparator;
lpszNameCopy[dwNameLength+1]=(TCHAR)_T('\0');
// Find The Name in the Query String
lpszIndex=_tcsstr(lpszStringCopy,lpszNameCopy);
delete lpszNameCopy;
// Error: The Name part of the Name value pair doesn't exist
if (!lpszIndex)
{
delete lpszStringCopy;
return (NULL);
}
// Increase the pointer to pass the Name and to Get the Value
lpszIndex+=_tcslen(lpszName)+1;
// Find the End of the Value by looking for the Delimiter
lpszEnd=_tcschr(lpszIndex,_T(cDelimiter));
// if we find a Demiliter set it as the end
if (lpszEnd)
(*lpszEnd)='\0';
// Remove the CGI Syntax
```

```
PreprocessString(lpszIndex);

// Calculate the Value Size

dwValueSize=_tcslen(lpszIndex);

lpszValue = new TCHAR[dwValueSize+1];

_tcscpy(lpszValue,lpszIndex);

delete lpszStringCopy;

return lpszValue;

}
```

Add the following FindNameValue() method to lst14_03.dll. Remember to define the method in the class definition. To test how this works, change Entrance to look for a name being passed in:

```
void  CLst14_03Extension::Entrance(CHttpServerContext* pCtxt,LPTSTR pszQuery)

{

ThreadPool();

(*pCtxt) << _T("<HTML><BODY>Name: ");

LPTSTR lpszValue;

if (lpszValue=FindNameValue(pszQuery,_T("Name"),_T('='),_T('&')))

{

(*pCtxt) << lpszValue;

delete lpszValue;

}

(*pCtxt) << _T("</BODY</HTML>");

LeavePool();

};
```

Notice that if FindNameValue() returns NULL, no memory has been allocated; otherwise, the caller, in this case Entrance(), needs to delete the memory allocated by FindNameValue().

Recompile and then execute lst14_03.dll through the browser, like this:

```
http://MYMACHINE/scripts/lst14_03/debug/lst14_03.dll?Name=Joe
```

where *MYMACHINE* is the name of your machine.

The browser should look like Figure 14.7.

Figure 14.7. Listing 14.3 running on a browser with the name of Joe.

Connecting to ODBC

One of the greatest advantages of using a Server Extension is that once it's loaded, it remains loaded in the cache. You can make a connection to an ODBC data source when the Server Extension is loaded and break the connection when the DLL is unloaded. This type of connection can be thought of as a constantly open pipe from the Server Extension to the data source. This makes for a huge performance advantage in querying data from a database because the most costly thing in making a small query to an ODBC data source is to open the connection. If the connection is already open when the request for information is made, the data comes back much more quickly.

The Internet Database Connector that comes with the IIS doesn't have this type of advantage. Every time a request is made to the IDC, the IDC opens a new connection to the database and closes it when it is done servicing the request. Server extensions that use the caching method described in the preceding paragraph can be much faster then the IDC.

Before discussing the code for this example, you need to create a database in SQL Server and a table called Surfers. After you've created a database in SQL Server, run this code in ISQL_w, and the table will be created for you:

```
CREATE TABLE dbo.Surfers (

Id int IDENTITY (1, 1) NOT NULL PRIMARY KEY,

Date datetime NULL DEFAULT GETDATE()

)
```

ISQL_w is an application that comes with SQL Server. After you create the table, create a data source entry in ODBC to point to the database and make the data source name "Surfers" so that it matches the example. If you don't know how to create an ODBC data source, refer to the section entitled "Creating a Data Source" in Chapter 16, "ISPI Internet Database Connector."

Create another Server Extension called lst14_04.dll by using the code in Listing 14.4. After you've created the Server Extension, you will have to add odbc32.lib to the Object|Library Modules Edit box under Build|Settings|Link so that the ODBC calls can link.

Listing 14.4 is the example of using ODBC to create a unique identifier.

Listing 14.4. lst14_04.cpp.

```
// LST14_04.CPP - Implementation file for your Internet Server

//    lst14_04 Extension

#include <afx.h>

#include <afxwin.h>
```

```
#include <afxisapi.h>

#include "resource.h"

#include "lst14_04.h"

//////////////////////////////////////////////////////////////////////

// The one and only CLst14_04Extension object

CLst14_04Extension theExtension;

CWinApp BugFix;

//////////////////////////////////////////////////////////////////////

// CLst14_04Extension implementation

CLst14_04Extension::CLst14_04Extension()

{

// Hard Coded Datasource, login, and password

m_csDSN="Surfers";

m_csUser="sa";

m_csPassword="";

InitODBC();

Connect();

}

CLst14_04Extension::~CLst14_04Extension()

{

Disconnect();

CleanUpODBC();

}

BOOL CLst14_04Extension::GetExtensionVersion(HSE_VERSION_INFO* pVer)

{

// Call default implementation for initialization

CHttpServer::GetExtensionVersion(pVer);
```

```
// Load description string

TCHAR sz[HSE_MAX_EXT_DLL_NAME_LEN+1];

ISAPIVERIFY(::LoadString(AfxGetResourceHandle(),

IDS_SERVER, sz, HSE_MAX_EXT_DLL_NAME_LEN));

_tcscpy(pVer->lpszExtensionDesc, sz);

return TRUE;

}

int CLst14_04Extension::CallFunction(CHttpServerContext* pCtxt

[ic:ccc],LPTSTR pszQuery, LPTSTR pszCommand)

{

int nRet=callOK;

ISAPIASSERT(pCtxt->m_pStream == NULL);

pCtxt->m_pStream = ConstructStream();

if (pCtxt->m_pStream == NULL)

nRet = callNoStream;

else

{

pCtxt->m_pStream->InitStream();

Entrance(pCtxt,pszQuery);

}

return(nRet);

}

void  CLst14_04Extension::Entrance(CHttpServerContext* pCtxt,LPTSTR pszQuery)

{

DWORD dwId;

dwId=UniqueId();

CString csOutput;
```

```
csOutput.Format(_T("Your Unique Id is: %d"),dwId);

(*pCtxt) << csOutput;

};

void CLst14_04Extension::InitODBC()

{

m_lpcsFlag=new CRITICAL_SECTION;

InitializeCriticalSection(m_lpcsFlag);

};

void CLst14_04Extension::CleanUpODBC()

{

DeleteCriticalSection(m_lpcsFlag);

delete m_lpcsFlag;

};

void CLst14_04Extension::Connect()

{

RETCODE rc;

rc=SQLAllocEnv(&m_henv);

if ((rc != SQL_SUCCESS) && (rc != SQL_SUCCESS_WITH_INFO))

ODBCError(SQL_NULL_HSTMT);

rc=SQLAllocConnect(m_henv, &m_hdbc);

if ((rc != SQL_SUCCESS) && (rc != SQL_SUCCESS_WITH_INFO))

ODBCError(SQL_NULL_HSTMT);

rc=SQLConnect(m_hdbc, (UCHAR FAR *)(LPCTSTR) m_csDSN,

SQL_NTS, (UCHAR FAR *)(LPCTSTR) m_csUser ,

[ic:ccc]m_csUser.GetLength(), (UCHAR FAR *)(LPCTSTR) m_csPassword,
m_csPassword.GetLength());

if ((rc != SQL_SUCCESS) && (rc != SQL_SUCCESS_WITH_INFO))

ODBCError(SQL_NULL_HSTMT);
```

```
};
void CLst14_04Extension::Disconnect()
{
SQLDisconnect(m_hdbc);
SQLFreeConnect(m_hdbc);
SQLFreeEnv(m_henv);
}
void CLst14_04Extension::ODBCError (HSTMT hstmt)
{
UCHAR FAR szSqlState[5];
UCHAR FAR szErrorMsg[80];
SWORD cbErrorMsgMax=80;
SWORD FAR *pcbErrorMsg= new SWORD;
SDWORD FAR *pfNativeError= new SDWORD;
RETCODE rc;
rc=SQLError(m_henv, m_hdbc, hstmt,
[ic:ccc] szSqlState, pfNativeError, szErrorMsg, cbErrorMsgMax, pcbErrorMsg);
if (!_tcscmp((char*)szSqlState,"08S01"))
{
// Bad Connection Let's try to recover!
ISAPITRACE("Bad Connection\n");
EnterCriticalSection(m_lpcsFlag);
Disconnect();
Connect();
LeaveCriticalSection(m_lpcsFlag);
}
ISAPITRACE1("SQL Error: %s\n",szErrorMsg);
```

```
delete pfNativeError;
delete pcbErrorMsg;
};
HSTMT CLst14_04Extension::GetStatement()
{
// The only reason that we are entering a critical section here
// is because the connection could be bad and we are trying to reconnect
// otherwise this isn't needed since ODBC is thread proof.
EnterCriticalSection(m_lpcsFlag);
HSTMT hstmt;
RETCODE rc;
rc=SQLAllocStmt(m_hdbc, &hstmt);
if ((rc != SQL_SUCCESS) && (rc != SQL_SUCCESS_WITH_INFO))
{
// Bad Connection Let's try to recover!
// The CriticalSection Prevents other threads
// From trying to get a statement with a bad connection
ISAPITRACE("Bad Connection\n");
EnterCriticalSection(m_lpcsFlag);
Disconnect();
Connect();
LeaveCriticalSection(m_lpcsFlag);
//Try Again
rc=SQLAllocStmt(m_hdbc, &hstmt);
if ((rc != SQL_SUCCESS) && (rc != SQL_SUCCESS_WITH_INFO))
{
// Only try to reconnect once
```

```
// Otherwise it is beyond our control

return(NULL);

}

}

LeaveCriticalSection(m_lpcsFlag);

return(hstmt);

}

void CLst14_04Extension::FreeStatement(HSTMT hstmt)

{

SQLFreeStmt(hstmt, SQL_DROP);

}

DWORD CLst14_04Extension::UniqueId()

{

DWORD Id=0;

HSTMT hstmt;

RETCODE rc;

SDWORD cbData;

if((hstmt=GetStatement())==NULL)

{

return(0);

}

CString SQLScript=

[ic:ccc]"INSERT Surfers DEFAULT VALUES SELECT Id=@@IDENTITY FROM Surfers WHERE
Id=@@IDENTITY ";

rc=SQLExecDirect(hstmt, (UCHAR FAR *) (LPCTSTR) SQLScript, SQL_NTS);

if ((rc != SQL_SUCCESS) && (rc != SQL_SUCCESS_WITH_INFO))

{
```

```
ODBCError(hstmt);

return(0);

}

rc = SQLMoreResults(hstmt);

if ((rc != SQL_SUCCESS) && (rc != SQL_SUCCESS_WITH_INFO))

{

ODBCError(hstmt);

return(0);

}

rc = SQLFetch(hstmt);

if ((rc != SQL_SUCCESS) && (rc != SQL_SUCCESS_WITH_INFO))

{

ODBCError(hstmt);

return(0);

}

rc=SQLGetData(hstmt, 1, SQL_C_SLONG, &Id, 0,&cbData);

if ((rc != SQL_SUCCESS) && (rc != SQL_SUCCESS_WITH_INFO))

{

ODBCError(hstmt);

return(0);

}

FreeStatement(hstmt);

return(Id);

};
```

The heading file, lst14_04.h:

```
// LST14_04.CPP - Implementation file for your Internet Server

//    lst14_04 Extension
```

```
#include <sql.h>
#include <sqlext.h>
class CLst14_04Extension : public CHttpServer
{
protected:
HENV m_henv;
HDBC m_hdbc;
LPCRITICAL_SECTION m_lpcsFlag;
Cstring m_csDSN;
Cstring m_csUser;
Cstring m_csPassword;
void InitODBC();
void CleanUpODBC();
void Connect();
void Disconnect();
void ODBCError(HSTMT hstmt);
public:
CLst14_04Extension();
~CLst14_04Extension();
BOOL GetExtensionVersion(HSE_VERSION_INFO* pVer);
int CallFunction(CHttpServerContext* pCtxt,LPTSTR pszQuery, LPTSTR pszCommand);
void  Entrance(CHttpServerContext* pCtxt,LPTSTR pszQuery);
HSTMT GetStatement();
void FreeStatement(HSTMT hstmt);
DWORD UniqueId();
};
```

After you've compiled the example, run it by typing

***MYMACHINE*\scripts\lst14_04\debug\lst14_04.dll**

where *MYMACHINE* is the name of your machine. The browser output should look something like Figure 14.8.

Figure 14.8. Request to Listing 14.4.

Notice that every time you refresh the browser, a unique ID is created in the database and displayed in the Web page. The ODBC connection is in the constructor of Server Extension, which gets called when the DLL loads. The ODBC connection is closed in the destructor. With the caching turned off for debugging, you do not benefit from the performance enhancement of a constantly open connection because the connection is being terminated with every load and unload of the DLL. Notice that the data source name, the user name, and the password and being statically defined in the constructor; this is to keep the example simple. Also, to simplify the example, thread pooling has been removed.

Losing the ODBC Connection

In a perfect world, a constantly open ODBC connection would not be a problem; however, this is not a perfect world. Because of failures in either hardware or software, the connection to ODBC can be lost. That's why it's important to write recovery software. Some recovery has been added to Listing 14.4; if the SQL Server connection is broken, the software tries to create a new connection. This keeps the Server Extension running and keeps your Web site up when the Server Extension encounters a minor problem. Notice that when the server extension is recovering, it enters a Critical Section, which makes sure that other threads entering the server extension do not disturb the recovery process.

Give Me a Cookie

Cookies are name/value pairs that are passed between the Web server and the Web browser. This is how a typical cookie implementation scheme works: If a cookie has been issued in the past, the browser passes that cookie to the server. The server extension then checks to see if a cookie has been passed to it. If the browser doesn't have that cookie, the server extension gives the browser a named cookie with a unique value. The next time the browser returns to the server, the browser tells the server what cookie it has gotten.

Cookies are great for keeping track of users who are traversing your Web space. They are also a good way to tell if a user is returning to your Web space.

With a little modification, Listing 14.4 can be used to issue cookies. You need to add a couple of methods, starting with GetServerVariable(), which retrieves variables passed from the server to the Server Extension. The server variable for the cookie example is HTTP_COOKIE, which is passed from the client to the server and then to the server extension. If the server extension has been visited before and a cookie has been assigned, this call will return a 'name=value' pair with the unique cookie identifier. Other server variables are listed in Table 13.2 in Chapter 13, "Windows CGI Scripting."

Here is the code for GetServerVariable(), method:

```
// The Caller must delete the memory, unless error in which case returns NULL

LPTSTR CLst14_04Extension::GetServerVariable(CHttpServerContext* pCtxt,

[ic:ccc] LPCTSTR pszVariableName)
```

```
{

LPVOID lpvBuffer=NULL;

DWORD dwSize=0;

pCtxt->GetServerVariable((LPTSTR)pszVariableName,NULL,(LPDWORD)&dwSize);

// Check to see if variable exists

if(dwSize==0)

return(NULL);

lpvBuffer=(LPVOID)new TCHAR[dwSize+1];

if (!(pCtxt->GetServerVariable((LPTSTR)pszVariableName,

[ic:ccc]lpvBuffer,(LPDWORD)&dwSize)))

{

delete lpvBuffer;

return(NULL);

}

if(dwSize==0)

{

delete lpvBuffer;

return(NULL);

}

return((LPTSTR)lpvBuffer);

};
```

You also have to add another method called FindNameValue(). FindNameValue() is used to separate the 'name=value' cookie pair if there is a cookie. Here is the code for FindNameValue():

```
// The Caller must delete the memory

LPTSTR CLst14_04Extension::FindNameValue(LPCTSTR lpszString,

[ic:ccc]LPCTSTR lpszName,TCHAR cSeparator,TCHAR cDelimiter)

{
```

```
LPTSTR lpszIndex;
LPTSTR lpszEnd;
DWORD dwValueSize=0;
LPTSTR lpszValue;
LPTSTR lpszStringCopy;
LPTSTR lpszNameCopy;
DWORD dwNameLength;
lpszStringCopy = new TCHAR[_tcslen(lpszString)+1];
_tcscpy(lpszStringCopy,lpszString);
lpszNameCopy = new TCHAR[_tcslen(lpszName)+2];
_tcscpy(lpszNameCopy,lpszName);
dwNameLength=_tcslen(lpszNameCopy);
lpszNameCopy[dwNameLength]=cSeparator;
lpszNameCopy[dwNameLength+1]=(TCHAR)_T('\0');
// Find The Name in the Query String
lpszIndex=_tcsstr(lpszStringCopy,lpszNameCopy);
delete lpszNameCopy;
// Error: The Name part of the Name value pair doesn't exist
if (!lpszIndex)
{
delete lpszStringCopy;
return (NULL);
}
// Increase the pointer passed the Name and Get to the Value
lpszIndex+=_tcslen(lpszName)+1;
// Find the End of the Value by looking for the Demiliter
lpszEnd=_tcschr(lpszIndex,_T(cDelimiter));
```

```
// if we find a Demiliter set it as the end

if (lpszEnd)

(*lpszEnd)='\0';

// Remove the CGI Syntax

PreprocessString(lpszIndex);

// Calculate the Value Size

dwValueSize=_tcslen(lpszIndex);

lpszValue = new TCHAR[dwValueSize+1];

_tcscpy(lpszValue,lpszIndex);

delete lpszStringCopy;

return lpszValue;

}
```

If a cookie is present, the cookie is checked to see if it is a contains a valid ID. The way the Server Extension checks this is too query the SQL Server and see if this id exists. To do this, use a method called ValidateId(). Here is the code for ValidateId().

```
BOOL CLst14_04Extension::ValidateId(DWORD dwId)

{

HSTMT hstmt;

RETCODE rc;

SDWORD cbData;

DWORD dwReturnId=0;

if((hstmt=GetStatement())==NULL)

{

return(FALSE);

}

CString SQLScript;

SQLScript.Format("SELECT Id FROM Surfers WHERE Id=%d",dwId);

rc=SQLExecDirect(hstmt, (UCHAR FAR *) (LPCTSTR) SQLScript, SQL_NTS);
```

```
if ((rc != SQL_SUCCESS) && (rc != SQL_SUCCESS_WITH_INFO))
{
ODBCError(hstmt);
return(FALSE);
}
rc = SQLFetch(hstmt);
if ((rc != SQL_SUCCESS) && (rc != SQL_SUCCESS_WITH_INFO))
{
ODBCError(hstmt);
return(FALSE);
}
rc=SQLGetData(hstmt, 1, SQL_C_SLONG, &dwReturnId, 0,&cbData);
if ((rc != SQL_SUCCESS) && (rc != SQL_SUCCESS_WITH_INFO))
{
ODBCError(hstmt);
return(FALSE);
}
FreeStatement(hstmt);
return(dwReturnId==dwId);
};
```

If a cookie doesn't exist or the cookie isn't valid, use the method UniqueId() to create a unique ID to use as a cookie. The server tells client what the cookie is by adding a line in the header of the page getting returned. When the client reads the page, it will assign the cookie. The next time the client returns to the Server Extension, that cookie will appear in the HTTP_COOKIE variable. The line inserted into the header not only contains the 'name=value' pair of the cookie, but also an expiration date for the cookie and a server path. After the expiration date, the client does not pass the cookie to the server anymore. For this example, the expiration date is set three years into the future (imagine what the Internet will be like in three years). Be careful in changing this because the day of the week is also included in the expiration date. The server path is set to the root of the server. This means that other dynamic Web pages in other directories on the same server can use the cookie you issued. You might consider changing the path to be local to your directory if others are using your server.

Note

Other Web servers cannot use the cookies you issue because the client returns a cookie only when it

returns to the site that issued the cookie.

The following code creates the cookie header and inserts it into the Web page header:

```
void CLst14_04Extension::CreateCookie (CHttpServerContext* pCtxt, DWORD dwId)
{
CString csCookie;
csCookie.Format(_T
("Set-Cookie: SurferId=%d; path=/; expires=Wednesday, 09-Nov-99 23:12:40 GMT \r\n"),
[ic:ccc]dwId);
EXTENSION_CONTROL_BLOCK *pECB =pCtxt->m_pECB;
DWORD dwHeaderSize = csCookie.GetLength();
pECB->ServerSupportFunction(pECB->ConnID,
 HSE_REQ_SEND_RESPONSE_HEADER,NULL, &dwHeaderSize,
[ic:ccc] (LPDWORD)(LPCSTR)csCookie);
}
```

Finally, the Entrance() method ties all the methods together in the right order:

```
void  CLst14_04Extension::Entrance(CHttpServerContext* pCtxt,LPTSTR pszQuery)
{
DWORD dwId=0;
LPTSTR lpszCookie;
LPTSTR lpszValue;
lpszCookie=GetServerVariable(pCtxt,"HTTP_COOKIE");
if (lpszCookie!=NULL)
{
if (lpszValue=FindNameValue(lpszCookie,"SurferId",'=',';'))
{
dwId=atoi(lpszValue);
```

```
delete lpszValue;

if (!ValidateId(dwId))

dwId=0;

}

delete lpszCookie;

}

if (!dwId)

{

dwId=UniqueId();

if (dwId)

{

CreateCookie(pCtxt,dwId);

}

}

CString csOutput;

csOutput.Format(_T("Your Unique Id is: %d"),dwId);

(*pCtxt) << csOutput;

};
```

Run Listing 14.4 in your browser. Notice that it assigns you a unique ID, and you continue to display that ID no matter how many times you refresh the browser. This example could be expanded to display the first time that the user viewed the page because the date the row was created is kept in the SQL Server.

Note

Some browsers don't support cookies. This means that when the server returns the Web page with a cookie in the header, the browser is not enabled to read the cookie or return the cookie. You need to write your code in such a way that both cookie-enabled browsers and cookie-disabled browsers can use your Web site. Make sure that you test your server extension with a cookie-disabled browser to insure that you Server Extensions work correctly.

Debugging Cookies

To test to make sure that your Server Extension is issuing a cookie correctly, you can look in the cookie file the browser creates. You can also erase the cookie it has given the browser from the cookie file. Erasing the cookie allows the Server Extension to issue you another one. To erase the cookie, you need to find the text file that the browser uses to store the cookies. Search your hard drive for files that begin with cookie. You will need to close the browser, open the file in a text editor such as Notepad, then erase the entry in the cookie file.

Summary

There are two main reasons for using Server Extensions instead of CGI scripts. One is that Server Extensions are faster because they share process space with the server, they are multithreaded, and they are cached in memory. Second, Server Extensions are easier to debug. Microsoft Developer Studio handles the debugging of DLLs nicely.

Server Extensions have advantages in accessing SQL Server data. Server Extensions are cached in memory, a constantly open connection can be made from the Server Extension to SQL Server. The open connection makes for faster data retrieval.

Chapter 17

ISAPI Filter Objects

ISAPI filter objects are extensions of the ISS Server. Filters add functionality to the server. In this way they are different from ISAPI server extensions that service requests from clients. ISAPI server filters affect all the web spaces of the server that the filter is on. Filters can be used to do security handling, log additional information, and modify output.

When the IIS is started, all the server filters referenced in the registry are loaded. When a request is made to the Server, filters that are designed to handle that specific request are called by the server. The following are different types of requests:

SF_NOTIFY_READ_RAW_DATA is called when the Server receives a request for raw data. In this case data is both the HTTP header and a body of text filling the request.

SF_NOTIFY_SEND_RAW_DATA is called when the Server is going to send raw data. In this case data is both the HTTP header and a body of text filling the request.

SF_NOTIFY_PREPROC_HEADERS is called when the Server is requested to preprocess the header files.

SF_NOTIFY_AUTHENTICATION is called when the Server is requested to authenticate the client.

SF_NOTIFY_URL_MAP is called when the Server receives a request to return the URL associated with the request.

SF_NOTIFY_LOG is called when the Server is going to log in to the database.

Server filters can be used for advanced logging of server activity. filters can modify the incoming or outgoing data, including adding your own server HTML tags. Server filters can also serve as an advanced means of authenticating the user.

Advanced Logging

You can use Server filter for advanced logging. The example chosen for advanced logging logs the number of bytes being requested from the server. This example is helpful if you are an Internet Service Provider and want to track which customers are using the most bandwidth. The example goes beyond a simple hit counter. Instead of hits, the server filter logs the size of the page and the size of the graphics on that page. The logging is done to SQL server, enabling various reports to be generated on the data.

Before you begin coding the example, you need to build the server filter with Microsoft's Extension wizard and learn how to debug the server filter with Microsoft's Developer Studio.

Creating a Filter Using Microsoft's Extension Wizard

In the Microsoft Developer's Studio there is a project wizard for creating server extensions. This ISAPI Extension Wizard can be used to create a filter extension. Use these steps:

1. 1. Open Microsoft's Developer Studio.
2. 2. From the menu bar, click on File|New.
3. 3. Select Project Workspace from the listbox and click OK.
4. 4. Go to the bottom of the Type listbox and choose ISAPI Extension Wizard.
5. 5. In the Location editbox, choose the default location c:\msdev\projects.

Note

With server extensions, you choose the script directory for the location of the server extension. This location enables you to run the DLL without copying it. However, server filters can be located anywhere and still run. A server filter does not need to be within the web space of the Web server.

---

1. 7. In the Name edit box, type the name of the project; for this example, enter **lst17_01**.
2. 8. Click Create.

Figure 17.1 is the first dialog box of the ISAPI Extension Wizard.

Figure 17.1. The Selection dialog box of the ISAPI Extension Wizard, step 1.

1. 9. Deselect the Generate a Server Extension object check box.

Note

Leaving MFC as a shared DLL increases the loading speed of your DLL. It also helps shared resources if more than one HTTP server extension or HTTP server filter is being run at the same time. Remember to copy mfc40.dll and msvcrt40.dll to your production server when you copy your release build of HTTP server filter.

---

1. 11. Click Next.

The following figure is the second dialog in the ISAPI Extension Wizard.

Figure 17.2. The Selection dialog box of the ISAPI Extension Wizard, step 2.

1. 12. For this example, leave the filter set to low priority. You also should select both secured and nonsecured ports.
2. 13. You need to choose the types of notification your filter will process. Because you are going to log the size of the data being sent, choose Outgoing raw data and headers.

Note

> You can choose more then one type, if you are going to do more then one type of processing. You also can make several DLLs, one for each type of processing. You benefit from putting all the filtering for your Web site in one DLL. The benefits include one process space, faster loading, and sharing of common procedures. The examples in this chapter, however, are divided—one filter per DLL. This filters them easier to read and understand.

14. Click Finish and OK.

Here is the code that should have been produced by the ISAPI Extension Wizard:

The Source for lst17_1.cpp:

```
// LST17_01.CPP - Implementation file for your Internet Server
//    lst17_01 Filter
#include <afx.h>
#include <afxwin.h>
#include <afxisapi.h>
#include "resource.h"
#include "lst17_01.h"
////////////////////////////////////////////////////////////////////////
// The one and only CLst17_01Filter object
CLst17_01Filter theFilter;
////////////////////////////////////////////////////////////////////////
// CLst17_01Filter implementation
CLst17_01Filter::CLst17_01Filter()
{
}
CLst17_01Filter::~CLst17_01Filter()
{
}
BOOL CLst17_01Filter::GetFilterVersion(PHTTP_FILTER_VERSION pVer)
{
// Call default implementation for initialization
CHttpFilter::GetFilterVersion(pVer);
// Clear the flags set by base class
pVer->dwFlags &= ~SF_NOTIFY_ORDER_MASK;
```

```
// Set the flags we are interested in

pVer->dwFlags |= SF_NOTIFY_ORDER_LOW | SF_NOTIFY_SECURE_PORT |
SF_NOTIFY_NONSECURE_PORT

| SF_NOTIFY_SEND_RAW_DATA | SF_NOTIFY_END_OF_NET_SESSION;

// Load description string

TCHAR sz[SF_MAX_FILTER_DESC_LEN+1];

ISAPIVERIFY(::LoadString(AfxGetResourceHandle(),

IDS_FILTER, sz, SF_MAX_FILTER_DESC_LEN));

_tcscpy(pVer->lpszFilterDesc, sz);

return TRUE;

}

DWORD CLst17_01Filter::OnSendRawData(CHttpFilterContext* pCtxt,

PHTTP_FILTER_RAW_DATA pRawData)

{

// TODO: React to this notification accordingly and

// return the appropriate status code

return SF_STATUS_REQ_NEXT_NOTIFICATION;

}

DWORD CLst17_01Filter::OnEndOfNetSession(CHttpFilterContext* pCtxt)

{

// TODO: React to this notification accordingly and

// return the appropriate status code

return SF_STATUS_REQ_NEXT_NOTIFICATION;

}

////////////////////////////////////////////////////////////////////

// If your extension will not use MFC, you'll need this code to make

// sure the extension objects can find the resource handle for the

// module.  If you convert your extension to not be dependent on MFC,

// remove the comments arounn the following AfxGetResourceHandle()

// and DllMain() functions, as well as the g_hInstance global.

/****
```

```
static HINSTANCE g_hInstance;

HINSTANCE AFXISAPI AfxGetResourceHandle()

{

return g_hInstance;

}

BOOL WINAPI DllMain(HINSTANCE hInst, ULONG ulReason,

LPVOID lpReserved)

{

if (ulReason == DLL_PROCESS_ATTACH)

{

g_hInstance = hInst;

}

return TRUE;

}

****/
```

The Header for Lst17_1.h:

```
#include <sqlext.h>

class CLst17_01Filter : public CHttpFilter

{

public:

CLst17_01Filter();

~CLst17_01Filter();

BOOL GetFilterVersion(PHTTP_FILTER_VERSION pVer);

DWORD OnSendRawData(CHttpFilterContext* pCtxt,

PHTTP_FILTER_RAW_DATA pRawData);

DWORD OnEndOfNetSession(CHttpFilterContext* pCtxt);

// TODO: Add your own overrides here

};
```

After you've created the DLL, compile it. Because the Extension Wizard creates nothing but the shell of the server filter, it will have no functionality.

All examples in this chapter start this way.

Debugging the HTTP Server Filter

The best way to debug the server filter is to run the filter through Microsoft Developer Studio. To do this, you need to be on the same machine as the one on which IIS is currently running. Make sure that this server is being used only by you; no other Web pages should be served from it. It is advisable also to run another copy of the Microsoft Developer Studio to make changes to and recompile your server filter. Use your browser on the same machine to call the Server.

Correct Permissions

Remember that the operating system believes that there are two users in the debugging scenarios. The first user, you, is debugging the IIS with the Microsoft Developer Studio. The second user, also you, is coming in from the network using a browser and calling the ISS Server. The first user has your login name and has to be an administrator. The second user is IUSR_*MACHINENAME* where *MACHINENAME* is the name of your machine. The second user gets set up by the IIS when it installs. This name can be changed in the Internet Service Manager; IUSR_*MACHINENAME* is the default.

In the debugging scenario, the second user permissions are adequate. However, special permissions are needed for the first user to run the IIS under Microsoft Developer Studio. If you are not in the administrator group on this machine, have an administrator make you one. Once you are an administrator, you need to give yourself a few more permissions. Use these steps:

1. 1. Make sure that you are logged on with administrator permissions.
2. 2. Choose Administrative Tools|User Manager for Domains.
3. 3. Make sure that User Manager is administrating the Machine that you are on.
4. 4. Select Policies|User Rights from the menu bar.
5. 5. Click on the check box in the bottom-right corner, which reads Show Advance User Rights.
6. 6. From the Right Select box, choose Act as part of the operating system.
7. 7. Click Add, and add yourself.
8. 8. Click OK.
9. 9. From The Right Select Box, choose act as part of Generate security audits.
10. 10. Click Add, and add yourself.
11. 11. Click OK.
12. 12. Click OK to exit User Rights.
13. 13. Exit User Manager.
14. 15. Log off and log back on.

No NT Security is set correctly so that you may run a service, like IIS, in the Debugger of Microsoft Developer Studio.

Disabling the IIS

To run the IIS on Microsoft Developer Studio, you need to stop running the default IIS. You can do this by going into the Internet Service Manager and stopping the server. You might also want to disable the server from restarting if you reboot the machine. Disabling the service allows you to come back to a development environment, instead of having to remember to stop the service every time you reboot. To do this, follow these steps:

1. 1. Go into Control Panel|Choose Services.
2. 2. Choose World Wide Web Publishing Service from the Services List.
3. 3. Click Startup.
4. 4. In the Startup dialog box, choose Manual, and click OK.
5. 5. Click Close.

6. 6. Exit the Control Panel.

Running the IIS in Microsoft Developer Studio

Now that you have adequate permissions, and have disabled the default IIS, you can run the IIS in Microsoft Developer Studio:

1. 1. Start another copy of Microsoft Developer Studio (besides the one open for compiling the server filter).
2. 2. From the menu bar, choose File|Open Workspace.
3. 3. From the Open Workspace dialog box's File of Type list, select Executable files (*.exe).
4. 4. Open the IIS executable c:\inetsrv\server\inetinfo.exe.
5. 5. From the menu bar of Microsoft Developer Studio, choose Build|Choose Settings.
6. 6. Click on the debug tab.
7. 7. Select General from the Category drop-down box.
8. 8. In the Program argument's edit box, type **-e W3Svc**.
9. 9. Select Additional DLLs from the Category drop-down box.
10. 10. In the Modules Box, choose the debug HTTP Server lst17_01.dll. It should be at c:\msdev\projects\lst17_01\debug\lst17_01.dll.
11. 11. Click OK and exit the Settings dialog box.
12. 12. From the menu bar, choose File|Save All.
13. 13. When asked Do you want to save project workspace information for C:\inetsrv\server\inetinfo.mdp, choose Yes. Then when you want to run the server filter in the Microsoft Developer Studio, you can just open C:\inetsrv\server\inetinfo.mdp, rather than executing all the previous steps.
14. 14. From the menu bar, choose Build|Execute inetinfo.exe.
15. 15. Every time you execute inetinfo.exe, you will be prompted with a Dialog that says the following:

```
'C:\inetsrv\server\inetinfo.exe' does not contain debugging information. Do you want
to continue?
```

1. Press Yes.

IIS is now running in the debugger of IIS. You can debug the server extension like any other DLL.

Configuring the IIS for Server Filter

For each server filter created, you need to tell the IIS that there is another filter to use. You do this using the following steps to modifying the register:

1. 1. Open the registry editor.
2. 2. Choose HKEY_LOCAL_MACHINE.
3. 3. Then select the SYSTEM\CurrentControlSet\Services\W3SVC\Parameters registry key.
4. 4. There is a value within Parameters called Filter DLLs. Add the newly-created DLL to the list of DLLs, making sure to separate each DLL by a comma. The default DLL, c:\inetsrv\Server\sspifilt.dll, should already be in the list. Adding Listing 17.1 should make the list look like this:

```
c:\inetsrv\Server\sspifilt.dll, c:\msdev\projects\lst17_01\debug\lst17_01.dll
```

Note

Adding the DLL to the registry is separate from adding the DLL in Microsoft's Developer Studio. Adding the

DLL to the registry tells the IIS to load the DLL and use it as a filter. Adding the DLL to the Additional DLL list in Microsoft's Developer Studio tells the debugger to load the symbols for debugging.

Filter Priority

More than one filter can be run for any particular notification. Each filter is loaded and runs in priority order. When the IIS starts, it reads the registry and loads the DLLs that are listed in the Filter DLLs key. For each DLL, it calls the GetFilterVersion() method, which returns the priority of the filter. This example uses the default priority, which is low. If there were more then one filter being used for a particular notification, the two DLLs should have different priorities to indicate which should run first and which should run second. If two DLLs have the same priority for the same notification, the Server executes the filters based on the Filter DLLs key in the registry. The first one in the list runs first.

Each filter has the capability of telling the server that it is the last filter that needs to be called or enabling the Server to execute the rest of the filters with that particular notification. If the filter wants to have the server stop executing filters that have its notification, it returns SF_STATUS_REQ_HANDLED_NOTIFICATION. If the filter wants the other filter "below" it (with lesser priority or the same priority but behind it in the list) to be called, the filter returns SF_STATUS_REQ_NEXT_NOTIFICATION.

Tip

If you keep all filtering in a single Server Filter, you do not need to deal with priorities or sequential executing filters.

Code for Advanced Logging

Now that you have created a lst17_01.dll with the ISAPI Extension Wizard, configured Microsoft's Developer Studio for debugging, and added the DLL to the registry so that IIS knows to load the filter, you are ready to program the Server Filter. The goal is to log the number of outgoing bytes for each page request to SQL Server.

Using ODBC

One of the greatest advantages of using a Filter Extension is that once it's loaded, it remains loaded in the cache. You can make a connection to an ODBC data source when the server extension is loaded and break the connection when the DLL is unloaded. This type of connection can be thought of as a constantly open pipe from the server extension to the data source. This offers a huge performance advantage in querying data from a database because the most costly thing in making a small query to an ODBC data source is opening the connection. If the connection is already open when the request for information is made, the data comes back much faster.

Before exploring the code for this example, you need to create a database in SQL Server and a table called TrafficLog. Once you have created a database in SQL Server, run the following code in ISQL_w and the table will be created for you:

```
CREATE TABLE dbo.TrafficLog (

TrafficLog_Id int IDENTITY NOT NULL PRIMARY KEY,

TrafficLog_Size int NULL DEFAULT 0,

TrafficLog_URL varchar (255) NULL ,

TrafficLog_Date datetime NULL DEFAULT GetDate()
```

```
)
```

ISQL_w is an application that comes with SQL Server.

After you create the table, create a data source entry in ODBC to point to the database; make the data source name TrafficLog so that it matches the example. If you don't know how to create an ODBC data source, refer to the section entitled Creating a Data Source in Chapter 18, "_____."

Add the methods below to the CLst17_01Filter class, they are used to connecting to ODBC:

```
void CLst17_01Filter::Connect()

{

RETCODE rc;

rc=SQLAllocEnv(&m_henv);

if ((rc != SQL_SUCCESS) && (rc != SQL_SUCCESS_WITH_INFO))

ODBCError(SQL_NULL_HSTMT);

rc=SQLAllocConnect(m_henv, &m_hdbc);

if ((rc != SQL_SUCCESS) && (rc != SQL_SUCCESS_WITH_INFO))

ODBCError(SQL_NULL_HSTMT);

rc=SQLConnect(m_hdbc, (UCHAR FAR *)(LPCTSTR) m_csDSN,

SQL_NTS, (UCHAR FAR *)(LPCTSTR) m_csUser , m_csUser.GetLength(),

(UCHAR FAR *)(LPCTSTR) m_csPassword, m_csPassword.GetLength());

if ((rc != SQL_SUCCESS) && (rc != SQL_SUCCESS_WITH_INFO))

ODBCError(SQL_NULL_HSTMT);

};
```

The Connect() method and the Disconnect() method are only called once, when the DLL is loading and unloading. They are used to connect and disconnect from the database. They are called from the destructor and the constructor of the CLst17_01Filter class, shown in Listing 17.1.

Note

With caching turned off for debugging, Connect() and Disconnect() will be called for every thread because the DLL is loading and unloading with each thread. You get the performance advantage only when the DLL is putting in production and the caching is turned on.

```
void CLst17_01Filter::Disconnect()

{

SQLDisconnect(m_hdbc);

SQLFreeConnect(m_hdbc);
```

```
SQLFreeEnv(m_henv);
}
void CLst17_01Filter::ODBCError (HSTMT hstmt)
{
UCHAR FAR szSqlState[5];
UCHAR FAR szErrorMsg[80];
SWORD cbErrorMsgMax=80;
SWORD FAR *pcbErrorMsg= new SWORD;
SDWORD FAR *pfNativeError= new SDWORD;
RETCODE rc;
rc=SQLError(m_henv, m_hdbc, hstmt, szSqlState,
pfNativeError, szErrorMsg, cbErrorMsgMax, pcbErrorMsg);
if (!_tcscmp((char*)szSqlState,"08S01"))
{
// Bad Connection Let's try to recover!
ISAPITRACE("Bad Connection\n");
EnterCriticalSection(m_lpcsFlag);
Disconnect();
Connect();
LeaveCriticalSection(m_lpcsFlag);
}
ISAPITRACE1("SQL Error: %s\n",szErrorMsg);
delete pfNativeError;
delete pcbErrorMsg;
};
HSTMT CLst17_01Filter::GetStatement()
{
// The only reason that we are entering a critical section here
// is because the connection could be bad and we are trying to reconnect
// otherwise this isn't needed since ODBC is thread proof.
```

```
EnterCriticalSection(m_lpcsFlag);
HSTMT hstmt;
RETCODE rc;
rc=SQLAllocStmt(m_hdbc, &hstmt);
if ((rc != SQL_SUCCESS) && (rc != SQL_SUCCESS_WITH_INFO))
{
// Bad Connection Let's try to recover!
// The CriticalSection Prevents other threads
// From trying to get a statement with a bad connection
ISAPITRACE("Bad Connection\n");
EnterCriticalSection(m_lpcsFlag);
Disconnect();
Connect();
LeaveCriticalSection(m_lpcsFlag);
//Try Again
rc=SQLAllocStmt(m_hdbc, &hstmt);
if ((rc != SQL_SUCCESS) && (rc != SQL_SUCCESS_WITH_INFO))
{
// Only try to reconnect once
// Otherwise it is beyond our control
return(NULL);
}
}
LeaveCriticalSection(m_lpcsFlag);
return(hstmt);
}
void CLst17_01Filter::FreeStatement(HSTMT hstmt)
{
SQLFreeStmt(hstmt, SQL_DROP);
};
```

The GetStatement() and FreeStatement() methods get a separate ODBC statement for each thread. All threads use the same connection, enabling the ODBC driver and SQL Server to handle the workload.

Tip

With only one open connection per filter, you need only one SQL Server license, because the licenses are issued by the number of connections.

You also need to modify the CLst17_01Filter class constructor and destructor use the code below:

```
CLst17_01Filter::CLst17_01Filter()

{

m_lpcsFlag=new CRITICAL_SECTION;

InitializeCriticalSection(m_lpcsFlag);

// Hard Coded Datasource, login, and password

m_csDSN="TrafficLog";

m_csUser="sa";

m_csPassword="";

Connect();

}

CLst17_01Filter::~CLst17_01Filter()

{

Disconnect();

DeleteCriticalSection(m_lpcsFlag);

delete m_lpcsFlag;

}
```

Make sure you declare the methods in the class definition and include the needed SQL Server variables:

```
class CLst17_01Filter : public CHttpFilter

{

protected:

HENV m_henv;

HDBC m_hdbc;

LPCRITICAL_SECTION m_lpcsFlag;

Cstring m_csDSN;
```

```
Cstring m_csUser;

Cstring m_csPassword;

HSTMT GetStatement();

void FreeStatement(HSTMT hstmt);

void ODBCError (HSTMT hstmt);

void Disconnect();

void Connect();

public:

CLst17_01Filter();

~CLst17_01Filter();

BOOL GetFilterVersion(PHTTP_FILTER_VERSION pVer);

DWORD OnSendRawData(CHttpFilterContext* pCtxt,

PHTTP_FILTER_RAW_DATA pRawData);

DWORD OnEndOfNetSession(CHttpFilterContext* pCtxt);

// TODO: Add your own overrides here

};
```

To link correctly you will have to add odbc32.lib to the Object|Library Modules edit box under Build|Settings|Link so that the ODBC calls can link.

Logging to SQL Server

To log to SQL Server, you need to modify SendRawData() which created by the ISAPI Extension Wizard:

```
DWORD CLst17_01Filter::OnSendRawData(CHttpFilterContext* pCtxt,

PHTTP_FILTER_RAW_DATA pRawData)

{

DWORD cbInData=pRawData->cbInData;

LPTSTR lpszURL;

if (lpszURL=GetServerVariable(pCtxt,"URL"))

{

Log(lpszURL,cbInData);
```

```
delete lpszURL;

}

// TODO: React to this notification accordingly and

// return the appropriate status code

return SF_STATUS_REQ_NEXT_NOTIFICATION;

}
```

Notice that the method GetServerVariable() is called on the variable "URL" so that you know which request is responsible for the data. The method Log() is called to do the actual logging of the data. GetServerVariable looks like this:

```
// The Caller must delete the memory, unless error in which case returns NULL

LPTSTR CLst17_01Filter::GetServerVariable(CHttpFilterContext* pCtxt,

 LPCTSTR pszVariableName)

{

LPVOID lpvBuffer=NULL;

DWORD dwSize=0;

pCtxt->GetServerVariable((LPTSTR)pszVariableName,NULL,(LPDWORD)&dwSize);

// Check to see if variable exists

if(dwSize==0)

return(NULL);

lpvBuffer=(LPVOID)new TCHAR[dwSize+1];

if (!(pCtxt->GetServerVariable((LPTSTR)pszVariableName,

lpvBuffer,(LPDWORD)&dwSize)))

{

delete lpvBuffer;

return(NULL);

}

if(dwSize==0)

{

delete lpvBuffer;

return(NULL);

}
```

```
return((LPTSTR)lpvBuffer);

};
```

The Log() method writes a row to the SQL Server every time it gets called. The Log() method looks like this:

```
void CLst17_01Filter::Log(LPCTSTR lpszURL, DWORD dwSize)

{

HSTMT hstmt;

RETCODE rc;

CString SQLScript;

if((hstmt=GetStatement())==NULL)

{

return;

}

SQLScript.Format(

"INSERT TrafficLog (TrafficLog_Size,TrafficLog_URL) VALUES (%d,'%s')",

dwSize,lpszURL);

rc=SQLExecDirect(hstmt, (UCHAR FAR *) (LPCTSTR) SQLScript, SQL_NTS);

if ((rc != SQL_SUCCESS) && (rc != SQL_SUCCESS_WITH_INFO))

{

ODBCError(hstmt);

goto End;

}

End:

FreeStatement(hstmt);

return;

}
```

Reviewing the Log

To review the log, open ISQL_w. Make sure the correct database is selected, and then run the following query:

```
SELECT SUM (TrafficLog_Size) BYTES, TrafficLog_URL
```

```
FROM TrafficLog

GROUP BY TrafficLog_URL
```

As an exercise, you can create a dynamic Web page that displays TrafficLog. You might want to wait until Chapter 18 to implement the page with the Internet Database Connector.

Using Filters to Change the Returning Data

Create a filter, using the ISAPI Extension Wizard in Microsoft Developer Studio, as you did for the preceding example. Call the new filter lst17_02. You still want to override the OnSendRawData() method. This time you look at the outgoing buffer and modify it. When your code sees the string segment <DATE>, you replace it with today's date. Then you can write regular HTML pages that have today's date in them. To do this, you need to add these two methods to CLst17_02Filter, shown in Listing 17.2.

```
LPTSTR CLst17_02Filter::Replace(LPTSTR lpszString,

LPTSTR lpszTarget, LPTSTR lpszReplacement)

{

LPTSTR lpszIndex;

LPTSTR lpszSegment;

LPTSTR lpszSegmentIndex;

LPTSTR lpszData;

DWORD dwIndexLength;

DWORD dwDataLength;

dwDataLength=_tcslen(lpszString)+_tcslen(lpszReplacement)+1;

lpszData=new TCHAR[dwDataLength];

_tcscpy(lpszData,lpszString);

while (lpszIndex=_tcsstr(lpszData,lpszTarget))

{

dwIndexLength=_tcslen(lpszIndex)+1;

lpszSegment=new TCHAR[dwIndexLength];

_tcscpy(lpszSegment,lpszIndex);

_tcscpy(lpszIndex,lpszReplacement);

lpszSegmentIndex=lpszSegment+_tcslen(lpszTarget);

_tcscat(lpszIndex,lpszSegmentIndex);

delete lpszSegment;
```

```
}
// If Needed Allocate some heap
if (_tcslen(lpszString)<_tcslen(lpszData))
{
delete lpszString;
lpszString = new TCHAR[_tcslen(lpszData)];
}
// Padd the end with spaces so that the
// Content Length stays the same
while (_tcslen(lpszString)>_tcslen(lpszData))
{
_tcscat(lpszData," ");
}
// Copy to Output
_tcscpy(lpszString,lpszData);
delete lpszData;
return lpszString;
}
DWORD CLst17_02Filter::OnSendRawData(CHttpFilterContext* pCtxt,
PHTTP_FILTER_RAW_DATA pRawData)
{
LPTSTR lpszData;
DWORD cbBufferSize=pRawData->cbInBuffer;
DWORD cbSize=pRawData->cbInData;
DWORD cbReplacementSize;
// This part creates the Date String
CTime tCurrent = CTime::GetCurrentTime();
LPTSTR lpszTime;
lpszTime = new TCHAR[6];
sprintf(lpszTime,_T("%d-%d"),tCurrent.GetMonth(),tCurrent.GetDay());
```

```
lpszData=new TCHAR[cbSize+1];

memcpy(lpszData,pRawData->pvInData,cbSize);

lpszData[cbSize]='\0';

lpszData=Replace(lpszData,_T("<DATE>"),(LPTSTR) lpszTime);

cbReplacementSize=_tcslen(lpszData);

// If the tag is smaller then the data that

// is replacing it them don't change the

// data.

if (cbReplacementSize <= (int)cbBufferSize)

{

_tcscpy((LPTSTR)pRawData->pvInData,lpszData);

}

delete lpszData;

delete lpszTime;

// TODO: React to this notification accordingly and

// return the appropriate status code

return SF_STATUS_REQ_NEXT_NOTIFICATION;

}
```

Remember to add the filter to the Filter DLLs key in the registry and to the additional DLLs in Microsoft Developer Studio. You are now ready to debug the code using Microsoft Developer Studio running the IIS.

This simple piece of HTML demonstrates the filter:

```
<HTML>

<BODY>

Current Date is: <DATE>

</BODY>

</HTML>
```

Note

The IIS allocates the space needed for the data being sent out. If you want to replace a piece of data with another one, you need to make sure that you have enough room. In other words, you must work within the limits of the buffer allocated by the IIS. The best way to do this is to create tags that are bigger then the data that you want to replace. For instance, the <DATE> tag has more characters in it than the biggest date, '12-31'.

Authentication

Server Filters can also be used for authentication of users who enter your Web space. Besides using the IIS authentication process, you can build your own by using Server Filters. Let's begin by creating a Server Filter using Microsoft Developer Studio's ISAPI Extension Wizard. Create it like the preceding example, but instead of choosing Outgoing raw data and headers, choose Client authentication request. Name the new filter lst17_03.

A Bug in Microsoft Developer Studio

When the ISAPI Extension Wizard creates the method shell for OnAuthentication, it looks like this:

```
DWORD OnAuthentication(CHttpFilterContext* pCtxt,

 PHTTP_FILTER_PREPROC_HEADERS pFiltInfo);
```

This isn't correct, however, and doesn't work. The method definition should look like this:

```
virtual DWORD OnAuthentication( CHttpFilterContext* pfc,

 PHTTP_FILTER_AUTHENT pAuthent );
```

You need to change this in both the class declaration and the method header in the example lst17_03.dll. This problem will be fixed in Microsoft Developer's Studio version 4.2.

Debugging Authentication

If you are debugging according to the instruction earlier in this chapter, you will notice that the IIS doesn't call OnAuthentication() every time you refresh the page. It calls OnAuthentication() only the first time in the session that you request a page. Even requesting a different page in the same Web space doesn't call OnAuthentication(). To call OnAuthentication() to test it, close the browser after the first authentication and reopen it to test again. This is true only for a successful request, however; if an error is returned by the OnAuthentication() process, it will be called again if the page is refreshed. Be careful of these circumstances when testing your Server Filter.

Authenticating on an IP Address

Listing 17.3, lst17_03.dll, checks the IP Address of the user making the request. If the IP address starts with 157.56, the user is authenticated; otherwise, the user is not. This type of authentication can be used to filter out people outside of a particular IP range. Add GetServerVariable() as a method. This is the second time that you have seen it. It was used in the first listing to retrieve the Server

variables passed in from the client. You use GetServerVariable() in Listing 17.3 to get the IP address of the user.

Listing 17.3.

```
// The Caller must delete the memory, unless error in which case returns NULL
LPTSTR CLst17_03Filter::GetServerVariable(CHttpFilterContext* pCtxt,
LPCTSTR pszVariableName)
{
LPVOID lpvBuffer=NULL;
DWORD dwSize=0;
pCtxt->GetServerVariable((LPTSTR)pszVariableName,NULL,(LPDWORD)&dwSize);
// Check to see if variable exists
if(dwSize==0)
return(NULL);
lpvBuffer=(LPVOID)new TCHAR[dwSize+1];
if (!(pCtxt->GetServerVariable((LPTSTR)pszVariableName,
lpvBuffer,(LPDWORD)&dwSize)))
{
delete lpvBuffer;
return(NULL);
}
if(dwSize==0)
{
delete lpvBuffer;
return(NULL);
}
return((LPTSTR)lpvBuffer);
};
```

You also need to modify OnAuthentication(). Notice the different return value and the header returned if a user is not authorized:

```
DWORD CLst17_03Filter::OnAuthentication(CHttpFilterContext* pCtxt,
PHTTP_FILTER_AUTHENT pHeaderInfo)
```

```
{

BOOL bNotAuthorized=TRUE;

LPTSTR lpszUser;

LPTSTR lpszIndex;

if(lpszUser=GetServerVariable(pCtxt,"REMOTE_ADDR"))

{

lpszIndex=_tcsstr(lpszUser,_T("."))+1;

lpszIndex=_tcsstr(lpszIndex,_T("."));

(*lpszIndex)=_T('\0');

bNotAuthorized=_tcscmp(lpszUser,"157.56");

delete lpszUser;

}

if (bNotAuthorized)

{

DWORD dwDataSize=_tcslen(_T("401 Unauthorized"));

pCtxt->ServerSupportFunction(  SF_REQ_SEND_RESPONSE_HEADER,

_T("401 Unauthorized"), NULL, &dwDataSize);

return SF_STATUS_REQ_FINISHED;

}

else

{

return SF_STATUS_REQ_NEXT_NOTIFICATION;

}

}
```

To test, remember that you need to add lst17_03.dll to the registry in the Filter DLLs key, and you also need to add lst17_03.dll to the additional DLLs list in Microsoft Developer Studio.

Summary

Filters enable advanced functionality to be added to the IIS Server. They enable the administrator of the Server to intercept a server request and reply in a way specific to the server.


TOC

Chapter 16

Internet Database Connector

by Wayne Berry

The *Internet Database Connector* is an ISAPI server extension DLL that Microsoft provides for connecting ODBC connections and the Web server. The DLL's name is httpodbc.dll, and it is referred to as the *IDC*. With the IDC, Web authors can create dynamic pages that connect to databases through ODBC without writing CGI scripts.

How the Internet Database Connector Works

When a URL with the extension idc is called on the server, the Web server turns control over to the IDC server extension. The IDC then loads the file related to that URL; once loaded, the IDC makes the calls described in that file to an ODBC data source. The information the data source returns is put into another file with the extension htx. The htx file acts like a template to display the information from the database. Once the information is formatted in the htx template, the template is passed back to the browser as an HTML page. Both the htx file and the idc file are text files and can be edited with a text editor. Each matching pair needs to be put into the scripts directory or a directory containing execute permissions.

Note

Because the idc file will contain a name and password to the database you're using, make sure that the directory in which the idc files are placed cannot be read. In other words, make sure that the directory has execute, but not read permission, with the IIS Manager.

Why Use the Internet Database Connector?

You can save yourself a lot of time when programming a Web site by using the IDC. Programming to an ODBC data source with a C++ script is somewhat tricky, and because the IDC is text based and doesn't need to be compiled, it is an easy way to retrieve information for a Web page. Second, all string and error are handle by the server extension, which allows the Web author to concentrate on the format and content of the Web page.

Working with the Examples

There are many ways to use the samples in this book, but the most beneficial way is to actually work through the examples. Because the SQL Server ODBC driver comes with IIS, this is the driver the examples will be based on. First, you must create a database in the SQL Server. Refer to the SQL Server instructions for creating a database. I have supplied a routine to create a table and some sample data in the newly created database. This routine can be run out of ISQL_w, a utility that comes with SQL Server. This routine (shown in Listing 16.1) creates a table called IDCSample. The table contains two attributes: Name and Age. The routine also fills in two rows of the table with 'John Doe' Age 24 and 'Jane Doe' Age 23.

Listing 16.1. Create the sample table.

```
CREATE TABLE IDCSample

(
```

```
Name varchar (20) NULL,

Age int NULL

)

GO

INSERT INTO IDCSample (Name,Age) VALUES ('John Doe',24)

INSERT INTO IDCSample (Name,Age) VALUES ('Jane Doe',23)
```

After you have created the database, the table, and the data, you need to make an ODBC data source connect to the database so that the IDC can connect to the database.

Creating a Data Source

The Internet Database Connector requires you to connect to a database through ODBC by using a data source. Name your example data source: WebSql. You can use the IDC to connect to any ODBC data source. Remember that when writing to other data sources, the SQL syntax may change from the example given throughout the chapter. To create an SQL Server data source, follow these steps:

1. 1. Open the control panel.
2. 2. Select the ODBC icon.
3. 3. Click the System DSN button in the bottom-right corner.
4. 4. Click the Add button.
5. 5. Select SQL Server from the Installed ODBC Drivers list.
6. 6. Enter the data source name; this is the name that the IDC will be using to connect your data source.
7. 7. Enter the Server name; if the Web server and the SQL Server are on the same machine, this selection should be (local).
8. 8. Enter the Database name.
9. 9. Exit by clicking OK and then Close and then Close again.

Tip

When creating a data source name for the Web server to connect to, make sure that it is a system data source name. The Web server can connect only to system data sources names.

After the data source is created, the IDC can be programmed.

An Overview of IDC Variables

There are four types of IDC variables that can be accessed in the htx, all of which are parsed into the htx file before it is sent back to the browser. These variables are referenced by adding <% before them and %> after them.

- Predefined variables: CurrentRecord and MaxRecords, which are used to monitor the activities of the details section.
- Attributes from the SQL statement: The name of the column returning from the SQL statement.
- Form parameters: These are passed to the idc but can be referenced in the htx by adding idc. in front of the name.
- Server variables: These are variables passed by the server to the IDC. Table 15.2 contains a complete list.

Programming the IDC

Let's begin by showing a query in Listing 16.2.

Listing 16.2. A simple query's .idc file.

```
Datasource: WebSql

Username: sa

Template: lst16_3.htx

SQLStatement:

+SELECT Name, Age

+FROM IDCSample
```

Notice that data source is the reference to the database. For these examples, we will be using the data source WebSql, which was created in the previous section "Creating a Data Source." The user name in this example is sa. We assume that the sa user name for SQL Server has no password. If there is a password, an extra line will need to be added that reads

```
Password: Wow
```

where Wow is the password. The Web server doesn't need to use the System Administrator Account (sa). The Web server can use any account that has permissions to execute the SQL statement in the idc file.

The URL to execute this IDC example is

```
http://MYMACHINE/scripts/lst16_2.idc
```

The IDC will load the file lst16_2.idc and then execute the SQL call in the SQL statement. It will then fill in the Template lst16_3.htx. The template to fill in must be referenced in the idc file.

The SQL statement for this example gets every Name and Age in the table.

Tip

To make sure that your SQL statement will work, use ISQL_w. ISQL_w is a query program that comes with SQL Server. Just copy the SQL statement from the idc file, paste it into ISQL_w, and remove the plus signs that are in front of each line. Make sure the correct database is selected and press the Run button. The result set should be the same as the result set for the idc.

Listing 16.3 provides a simple query to demonstrate the Internet database connector.

Listing 16.3. A simple query's .htx file.

```
<HTML>

<BODY>

<%begindetail%>

Name: <%Name%> Age: <%Age%><BR>

<%enddetail%>

</BODY>

</HTML>
```

The htx file greatly resembles an HTML file. The difference is that the htx file can use special tags that are parsed out before the page is sent to the server. These IDC tags are not like the regular HTML tags. HTML tags are send to the client where they are used for formatting. IDC tags are used by the IDC so that it knows where to insert information from the query. The page that is sent to the client doesn't have any IDC tags in it. The Internet database connector tags used in this example are

```
<%Name%>

<%Age%>

<%begindetail%>

<%enddetail%>
```

All IDC tags start with <% and end with <%. When the IDC makes the SQL call in the example, it will come back with a result set that contains two rows. It then iterates between the <%begindetail%> and the <%enddetail%> IDC tags for each row in the result set. When the IDC reaches <%Name%>, it will parse out <%Name%> and replace it with the actual Name attribute in the row. This is the same for <%Age%> except it will replace <%Age%> with the <%Age%> attribute.

The section of the htx page that looks like this:

```
<%begindetail%>

Name: <%Name%> Age: <%Age%><BR>

<%enddetail%>
```

will look like this:

```
Name: John Doe Age: 24<BR>

Name: Jane Doe Age: 23<BR>
```

when it is sent to the client from the IDC.

The result from running the example looks like Figure 16.1.

Figure 16.1. The result of a simple query.

The first result set actually comes back before the IDC reaches the <%begindetail%> tag. This is important to know in case you want to use some part of the first row returned for the title of the HTML page and also have the row in a table with the other rows later on in the page. Listings 16.4 and 16.5 illustrate this.

Listing 16.4. An experiment with the <%begindetail%> tag in the .idc file.

```
Datasource: WebSql

Username: sa

Template: lst16_5.htx

SQLStatement:

+SELECT Name, Age

+FROM IDCSample
```

Listing 16.5. An experiment with the <%begindetail%> tag in the .htx file.

```
<HTML>
```

```
<BODY>

The Name Before the begindetail tag: <%Name%> <BR>

<HR>

<%begindetail%>

The Name Inside: <%Name%><BR>

<%enddetail%>

<HR>

The Name After the begindetail tag: <%Name%> <BR>

</BODY>

</HTML>
```

The URL to execute this IDC example is

```
http://MYMACHINE/scripts/lst16_4.idc
```

where MYMACHINE is your machine name. (See Figure 16.2.)

Figure 16.2. The results of an experiment.

Notice that the first row in the table can be accessed before the <%begindetail%> tag. This trick increases the flexibility of the IDC.

Selecting Rows to View

Viewing all the rows in a particular table is useful, but it is equally useful to be able to select certain rows from the table and to view them.

We can send parameters from an HTML form to the idc file. The idc file then uses these parameters to construct an SQL statement to the database. (See Listing 16.6.)

Listing 16.6. The HTML selection form.

```
<HTML>

<BODY>
```

```
<FORM ACTION="../scripts/lst16_6.idc" METHOD=POST>

<SELECT NAME="Age">

<OPTION>24

<OPTION>23

<OPTION>54

<INPUT TYPE=SUBMIT>

</FORM>

</BODY>

</HTML>
```

lst16_6.htm will have to be put into the wwwroot directory where it can be read. The action of the form calls the lst16_7.idc, which must be in the scripts directory where it can be executed. You might have to change the path in the ACTION attribute if you have your scripts directory set up differently than the default.

Lst16_6.htm should look like Figure 16.3.

Figure 16.3. Listing 16.6's HTML displayed

Notice that the form in p16_6.htm sends a parameter called Age to lst16_7.idc. We can use the Age parameter to select only those people that are that age. Listing 16.7 shows an .idc file that does this:

Listing 16.7. The idc file.

```
Datasource: WebSql

Username: sa

Template: lst16_3.htx

SQLStatement:

+SELECT Name, Age

+FROM IDCSample

+WHERE Age=%Age%
```

Notice that the idc is using the template lst16_3.htx, the same file that we used for lst16_2.idc. This is a good example of recycling the files and works perfectly fine.

After the IDC loads the idc file from the hard drive, it substitutes the %Age% variable with the Age that is sent in from the form. It then executes the query on SQL Server. If 24 is submitted as the age, the SQL query looks like

this:

```
+SELECT Name, Age

+FROM IDCSample

+WHERE Age=24
```

Because the Age attribute in the database is an integer, it doesn't need to be encapsulated in single quotes. If Age were a varchar or text attribute, the WHERE clause in the idc file would look like this

```
+WHERE Age='%Age%'
```

and the actual query would look like this

```
WHERE Age='24'
```

if 24 were selected.

If 24 is selected from lst16_6.htm using the example database, the result looks like Figure 16.4.

Figure 16.4.The result of selecting 24 from lst16_6.htm.

If 56 is selected from lst16_6.htm using the example database, the page will be empty because there is no one in the database with an age of 56.

Running Two SQL Statements in the Same idc File

The IDC is only able to handle one result set for each page it displays. This means that if you want to run two SQL statements in the idc file, there must be only one result set returned. The first result set listed in the idc file will be the one that the IDC uses. SQL statements like INSERT and DELETE return empty result sets. If you wanted to INSERT a person into the example table and SELECT all the people in the same table to display, you could not do this:

```
Datasource: WebSql

Username: sa

Template: lst16_3.htx

SQLStatement:
```

```
+INSERT IDCSample (Name,Age) VALUES ('Joe Junior',3)

+SELECT Name, Age

+FROM IDCSample
```

The problem with doing this is that the IDC will display the result set of the INSERT that is blank and will not display the result set of the SELECT statement. The reverse will work, however.

```
Datasource: WebSql

Username: sa

Template: lst16_3.htx

SQLStatement:

+SELECT Name, Age

+FROM IDCSample

+INSERT IDCSample (Name,Age) VALUES ('Joe Junior',3)
```

The problem with this is that the SELECT statement doesn't include Joe Junior because it isn't inserted into the table until after the SELECT statement is run.

To accomplish this task, you need to merge both SQL statements into a stored procedure on the SQL Server. Listing 16.8 is sample code that is run in the ISQL_w to create the stored procedure that we will be using.

Listing 16.8. A stored procedure for inserting and selecting.

```
CREATE PROCEDURE LST16_8 AS

INSERT INTO IDCSample (Name,Age) VALUES ('Joe Junior',3)

SELECT Name, Age

FROM IDCSample
```

Now all we have to do is write an SQL statement to call the stored procedure from the idc file. (See Listing 16.9.)

Listing 16.9. The idc file.

```
Datasource: WebSql

Username: sa

Template: lst16_3.htx
```

```
SQLStatement:

+EXECUTE Lst16_8
```

Run lst16_9.idc by typing its URL into the browser. Notice that Joe Junior is inserted in the table and displayed. Now refresh the URL server times and watch Joe Junior data being inserted for every refresh.

Notice that out of the two SQL statements, one returns no rows and the other returns many rows. You cannot have a stored procedure that has two SELECT statements in it. This trick will work with any number of SQL statements as long as there is only one SELECT statement.

Sending Parameters to Stored Procedures

If a form is sending the idc file parameters from input tags, the idc file must be able to pass these parameters to the stored procedure. The stored procedure must be able to take parameters. The stored procedure in Listing 16.10 is a variant of Listing 16.8.

Listing 16.10. A second stored procedure.

```
CREATE PROCEDURE LST16_10

@Name varchar(20),

@Age int

AS

INSERT INTO IDCSample (Name,Age) VALUES (@Name,@Age)

SELECT Name, Age

FROM IDCSample
```

Run Listing 16.10 in ISQL_w so that the stored procedure is created.

Now an idc file can be created that calls the stored procedure. (See Listing 16.11.)

Listing 16.11. Calling Lst16_10 from an .idc file.

```
Datasource: WebSql

Username: sa

Template: lst16_3.htx

SQLStatement:
```

```
+EXECUTE Lst16_10 @Name="%Name%", @Age=%Age%
```

Listing 16.12 shows an HTML page that calls the idc file sending the Name and Age that is entered into the page.

Listing 16.12. The .htx file to display the date.

```
<HTML>

<BODY>

<FORM ACTION="../scripts/lst16_11.idc" METHOD=POST>

Name : <INPUT TYPE=TEXT NAME="Name"><BR>

Age: <INPUT TYPE=TEXT NAME="Age"><BR>

<INPUT TYPE=SUBMIT>

</FORM>

</BODY>

</HTML>
```

Remember that you will need to save the .htm file in a directory with read permission, not the same directory as the .idc file. You might have to modify the form action in the .htm file based on where the .idc file is located.

If you are a high-level language programmer, you probably understand about variables, variable passing, and variable casting. If not, this discussion might not make a lot of sense, but it is important.

Notice in the EXECUTE statement that %Name% is wrapped in double quotes; this is because Name is a varchar and can contain spaces. Also notice that in the HTML page, both variables are of type text. They are represented as strings and are passed as strings to the idc file. At the idc file's level, they are still strings and are parsed into the SQL statement as strings. Not until the SQL Server's query engine reads the text-based SQL statement are the variables divided into integers and varchars. When the SQL Server returns the information, the IDC converts the attributes of the rows back to text and parses them into the htx file. The only time you have to worry about variable passing is when you try to insert something into SQL Server and when using the if-then-else clause of the idc. The if-then-else clause will be discussed in another chapter.

Try the htm file; type a name and an age.

Common Bugs with the IDC

The previous example (consisting of Listings 16.10 through 16.12) is riddled with bugs. They are put there on purpose to illustrate some points about the IDC. The bugs come when the users of the HTML page enter in bad data or unexpected data. Remember on the Internet, anyone can reach your HTML page, and anyone can enter data.

Try entering John and Thirty for the name and age. You should get a screen that looks like Figure 16.5.

Figure 16.5. Browser view after inputting Thirty.

This error occurs when SQL server tries to convert thirty to an integer and it can't do it. There are two solutions to this problem. One is to offer a selection box on the HTML page that is full of all the possible ages. The other is to make the Age field a varchar so that anything can be entered in. The last fix means that the users could enter anything; for instance, None of your business. You would then need to sort out the bad data at a later time. If your database is large, extra work is involved to remove the bad data.

Try entering Suzzy and 3.5 for the name and age. Notice that the 3.5 is rounded to 3. This happens because SQL Server rounds the fraction to an integer when it is inserted in the database.

Try entering for the name John's Kid and an age of 5. The entry in the database will look like this: John"s Kid.

The error occurs because the SQL Server thinks that you're referring to the end of a string when a ‘ is typed. There is very little we can do about this error, although entering John's Kid works correctly.

Try entering for the name Joseph 'Joe' Doe and an Age of 102. The entry in the SQL Server looks like this: "Joseph `Joe' Doe". The problem here is that the SQL Server changes the first single quote into a `.

Try entering for the name Theodore Roosevelt, Jr. and an Age of 30. Notice that the entry is truncated to Theodore Roosevelt. This is caused by the fact that we allowed the Name to be only 20 characters in SQL Server. When the INSERT command occurs, the name is truncated. The solution to this problem is to change the <INPUT> tag to have a maximum length. This can be set in the HTML page like this:

```
Name : <INPUT TYPE=TEXT NAME="Name" MAXLENGTH=20><BR>
```

Aggregate Functions Without Stored Procedures

In the previous section, you learned how to have two SQL statements called from an idc file. The only limitation is that both cannot be SELECT statements. There is a workaround to this problem. If the second SELECT is programmed correctly and we modify the htx file, we can have two select statements in one idc file. This workaround is most useful when used with aggregate functions. In our example reports, it would be interesting to see all the people in the table and their average age. In standard SQL, this would require two SELECT statements:

```
SELECT Name, Age

FROM IDCSample

SELECT AVG(Age)

FROM IDCSample
```

These two SQL statements return two result sets; however, if we rewrite the SQL statements, they will return only one result set. (See Listing 16.13.)

Listing 16.13. The idc file.

```
SELECT Age, Name

FROM IDCSample

UNION

SELECT AVG(Age) Age, 'zzzzzAverage' Name

FROM IDCSample

ORDER BY Name
```

The UNION operator was used to bind the two SELECT statements, which causes the output to be only one result set. Run this routine in ISQL_w. Notice that the result set comes back with all the names and ages in order, and the very last row is 'zzzzzAverage' with the Age column being the Average Age. Now let's put the query into an idc file, shown in Listing 16.14.

Listing 16.14. The Average idc.

```
Datasource: WebSql

Username: sa

Template: lst16_15.htx

SQLStatement:

+SELECT Age, Name

+FROM IDCSample

+UNION

+SELECT AVG(Age) Age, 'zzzzzAverage' Name

+FROM IDCSample

+ORDER BY Name
```

We will have to do something special with the htx file to divide out the Average from the rest of the rows. (See Listing 16.15.)

Listing 16.15. The Average htx.

```
<HTML>
```

```
<BODY>

<%begindetail%>

<%if Name EQ "zzzzzAverage" %>

Average: <%Age%>

<%else%>

Name: <%Name%> Age: <%Age%><BR>

<%endif%>

<%enddetail%>

</BODY>

</HTML>
```

There are three new IDC tags in this htx.

```
<%if%>

<%else%>

<%endif%>
```

These new tags create an if-then-else clause in the htx. This clause is resolved during runtime on the server side. We will talk more about this in the next section. The output to the client looks like this:

```
<HTML>

<BODY>

Name: Jane Doe Age: 23<BR>

Name: Joe Junior Age: 3<BR>

Name: John Doe Age: 24<BR>

<B>Avergage: 17</B>

</BODY>

</HTML>
```

Because we told the SQL statement to order by the Name attribute and we named the Average "zzzzzAverage", Average ends up at the bottom of the page where we want it. (See Figure 16.6.)

Figure 16.6. Browser view after running Listing 16.14.

It is possible to have more than one SELECT statement in a single idc file if you bind the SELECT statements with the UNION operator.

IDC's if-then-else

Let's take a closer look at the if-then-else expression used in the previous section as one of the sample problems.

```
<%if Name EQ "zzzzzAverage" %>
```

When comparing strings, the constant strings need to have double quotes surrounding them, as the "zzzzzAverage" has. Also, the variables coming back from SQL Server do not have to have the <% %> marks around them like the rest of the page. The EQ operator with the string comparison is case-sensitive. The other operators that can be used in the expression are shown in Table 16.1.

Table 16.1. Table of operators for the Internet Database Connector.

Operator Description EQ Equals operator LT Less Than GT Greater Than CONTAINS If the left side is in the right side, then true

Notice that the AND, OR, or NOT operators aren't there. To work around not having a NOT operator, just remove the NOT and flip the false section with the true section. If this were your section of code

```
<% if NOT Name EQ "Joe" %>

<H1>Expression is True</H1>

<%else%>

<H1>Expression is False</H1>

<%endif%>
```

you would have to change it because there is no NOT operator.

```
<% if Name EQ "Joe" %>

<H1>Expression is False</H1>
```

```
<%else%>

<H1>Expression is True</H1>

<%endif%>
```

You can't nest an if-then-else clause. The following example doesn't work:

```
<% if Name EQ "Joe" %>

<% if Name EQ "Smith" %>

<H1>It's Joe Smith</H1>

<%else%>

<H1>It's only a Joe</H1>

<%endif%>

<%else%>

<H1>This is not Joe</H1>

<%endif%>
```

Using Dates with the IDC

The reason that dynamic pages are created is because something is changing on a regular basis, and you want to be able to reflect that in your Web page. What is changing could be any of the following: the user, the data in the database, or the date. Let's look at an example of the latter. Suppose that you had a table of names and birthdays and you wanted the Web page to show all the birthdays for today. First, let's create a table in the database that has this information. Use the following routine in the ISQL_w to create the table in our example database WebSql:

```
CREATE TABLE IDCBirthday

(

Name varchar (20) NULL,

Birthday datetime

)
```

```
GO

INSERT INTO IDCBirthday (Name,Birthday) VALUES ('John Doe','4 Oct 1971')

INSERT INTO IDCBirthday (Name,Birthday) VALUES ('Jane Doe','9 Nov 1967')
```

This routine creates a table with Name and Birthday attributes, along with two rows. Now let's create an idc file (see Listing 16.16) that finds out if anyone has a birthday today.

Listing 16.16. Comparing dates with an .idc file.

```
Datasource: WebSql

Username: sa

Template: lst16_16.htx

SQLStatement:

+SELECT Name, Birthday

+FROM IDCBirthday

+WHERE CONVERT(varchar (5),GetDate(),101) = CONVERT(varchar (5),Birthday,101)
```

We will also need a template to display the birthdays. (See Listing 16.17.)

Listing 16.17. Displaying dates in the .htx file.

```
<HTML>

<BODY>

<%begindetail%>

Name: <%Name%> Age: <%Birthday%><BR>

<%enddetail%>

</BODY>

</HTML>
```

In the Where line of the SQL statement, we have converted Birthday to the first five characters. The first five characters are the month in two digits, the day in two digits, and the delimiter. We do this so that we aren't comparing the year or the time of the datetime variable. SQL Server's GetDate() function is used to get the current date; it is also truncated to the first five characters. You need to use SQL Server's Convert() function for all datetime variables because the IDC has no formatting conventions.

Because this query doesn't take any parameters, you can run it right from the browser address box. Open the idc from your Web browser. In most cases, you should see a blank screen. This is because of the day you are running it. Is it the fourth of October or the ninth of November? Try adding to the database another row that has today's date. Use this routine in the ISQL_w to do this:

```
INSERT INTO IDCBirthday (Name,Birthday) VALUES ('Joe Jr.',GetDate())
```

Now open the idc in the browser again. Notice that the date and time to the millisecond are displayed for the birthday. The date and time to the millisecond is the default return for SQL Server. To get the data we want, we will have to convert the Birthday to a varchar using the CONVERT function. Replace the SELECT line in Listing 16.17 with

```
+SELECT Name, CONVERT(varchar (5),Birthday,101) Birthday
```

Now open the idc in the browser again. Notice that Birthday is now just the month and the day.

CurrentRecord and MaxRecords

CurrentRecord and MaxRecords are variables that are built into the IDC. They can be used in the if-then-else expression in the htx file. You cannot view either CurrentRecord or MaxRecords in the page. Neither <%CurrentRecord%> nor <%MaxRecords%> tags work. MaxRecords is set to the value that MaxRecords is defined as in the idc file. CurrentRecord is a count of the rows that have come down from the SQL Server. For example, before the <%begindetail%> tag and for the first row in the SQL statement, CurrentRecord equals 0. When <%enddetail%> is reached and the IDC goes back to the <%begindetail%> tag for another row, CurrentRecord is incremented by one. If no rows are returned from the SQL statement, the detail section is skipped and CurrentRecord equals zero after the <%enddetail%>, also. Suppose that in the preceding example, we wanted to display the message "No birthdays today" instead of having a blank page. We could use code similar to Listing 16.18.

Listing 16.18. An example of CurrentRecord.

```
<HTML>

<BODY>

<%begindetail%>

Name: <%Name%> Age: <%Birthday%><BR>

<%enddetail%>

<%if CurrentRecord EQ 0%>
```

```
No birthdays today

<%endif%>

</BODY>

</HTML>
```

To test this, modify lst16_16.idc to use the template lst16_18.htx. You will also have to remove any birthdays that happened today. Use this SQL routine in ISQL_w:

```
DELETE IDCBirthday

WHERE CONVERT(varchar (5),GetDate(),101) = CONVERT(varchar (5),Birthday,101)
```

Now open lst16_17.idc in your browser, and you should see something similar to Figure 16.7.

Figure 16.7. No Birthdays.

This works because if no rows are returned, CurrentRecord still equals zero after <%enddetail%>.

Another good use of the CurrentRecord variable is to add a title row to an HTML table. Listing 16.19 shows an example of such a table.

Listing 16.19. Title rows using CurrentRecord.

```
<HTML>

<BODY>

<TABLE BORDER=1>

<%begindetail%>

<%if CurrentRecord EQ 0%>

<TR>

<TD><B>Name</B></TD>

<TD><B>Birthday</B></TD>

</TR>

<%endif%>

<TR>
```

```
<TD><%Name%></TD>

<TD><%Birthday%></TD>

</TR>

<%enddetail%>

</TABLE>

<%if CurrentRecord EQ 0%>

No birthdays today

<%endif%>

</BODY>

</HTML>
```

Here we have taken the data coming back from the SQL statement and put it into a table.

This section of code

```
<%if CurrentRecord EQ 0%>

<TR>

<TD><B>Name</B></TD>

<TD><B>Birthday</B></TD>

</TR>

<%endif%>
```

displays bold titles if there is a least one row and that row is the first row returned.

Notice that the table tags are outside of the <%begindetail%> <%enddetail%> tag. If they were inside, there would be one table for every SQL row instead of one HTML row for every SQL row. The only problem is that if no rows are returned from SQL Server, there will be an empty table returned to the browser. This is not really a big problem because empty tables are not displayed on most browsers.

To test this, modify lst16_16.idc to use the template lst16_19.htx. You will also have to add a birthday to view by using Listing 16.18.

Now open lst16_16.idc in your browser. (See Figure 16.8.)

Figure 16.8. Birthday table.

The MaxRecords variable is best used when there are large numbers of rows returned from the SQL Server and

you want to display only a few of them at a time. For the examples of MaxRecords, we will need another table. Use Listing 16.20 in ISQL_w to create another table under the WebSql database.

Listing 16.20. Table creating script for IDCCompany table.

```
CREATE TABLE IDCCompany

(

Id int NOT NULL IDENTITY PRIMARY KEY,

Name varchar (20) NULL,

Location varchar(2) NULL,

)

GO

INSERT INTO IDCCompany (Name,Location) VALUES ('Company A','WA')

INSERT INTO IDCCompany (Name,Location) VALUES ('Company B','OR')

INSERT INTO IDCCompany (Name,Location) VALUES ('Company C','WA')

INSERT INTO IDCCompany (Name,Location) VALUES ('Company D','OR')

INSERT INTO IDCCompany (Name,Location) VALUES ('Company E','WA')

INSERT INTO IDCCompany (Name,Location) VALUES ('Company F','WA')

INSERT INTO IDCCompany (Name,Location) VALUES ('Company G','OR')

INSERT INTO IDCCompany (Name,Location) VALUES ('Company H','WA')

INSERT INTO IDCCompany (Name,Location) VALUES ('Company I','WA')

INSERT INTO IDCCompany (Name,Location) VALUES ('Company J','WA')

INSERT INTO IDCCompany (Name,Location) VALUES ('Company K','WA')

INSERT INTO IDCCompany (Name,Location) VALUES ('Company L','WA')

INSERT INTO IDCCompany (Name,Location) VALUES ('Company M','WA')

INSERT INTO IDCCompany (Name,Location) VALUES ('Company N','WA')
```

The table created has an ID that is both an identity and a primary key. This ID will be used to track which records have been viewed already.

Now let's create an idc file (see Listing 16.21) that calls the newly created table.

Listing 16.21. Selecting from the IDCCompany table.

```
Datasource: WebSql

Username: sa

Template: lst16_25.htx

MaxRecords: 10

SQLStatement:

+SELECT Id, Name, Location

+FROM IDCCompany

+WHERE IDCCompany.Id >= %Id% AND IDCCompany.Location = '%LOCATION%'
```

Notice that MaxRecords has been defined as 10 in the idc file.

The matching template using MaxRecords is shown in Listing 16.22.

Listing 16.22. Displaying from the IDCCompany table.

```
<HTML>

<BODY>

<TABLE BORDER=1>

<%begindetail%>

<%if CurrentRecord EQ 0%>

<TR>

<TD><B>Name</B></TD>

<TD><B>Location</B></TD>

</TR>

<%endif%>

<TR>

<TD><%Name%></TD>
```

```
<TD><%Location%></TD>

</TR>

<%enddetail%>

</TABLE>

<%if 'Id' EQ '0' %>

<%else%>

<A HREF=lst16_24.idc?Id=<%Id%>&Location=<%Location%>>Next 10 Records</A>

<%endif%>

<%if CurrentRecord EQ 0%>

No Matches

<%endif%>

</BODY>

</HTML>
```

If, after the <%enddetail%>, an ID is present, we know that <%enddetail%> ended because the maximum number of records was reached. Because there might be more records, we put in an anchor to call the same idc again to get the next set of records. If an ID isn't present, we know that we have all the records. The ID would not be available if MaxRecords were not set in the idc.

```
<%if CurrentRecord EQ MaxRecords%>

<A HREF=lst16_24.idc?Id=<%Id%>&Location=<%Location%>>Next 10 Records</A>

<%endif%>
```

Now create an htm file (see Listing 16.23) to call the idc and save the htm file in the wwwroot directory.

Listing 16.23. HTML to call lst16_24.idc.

```
<HTML>

<BODY>

<FORM ACTION="../scripts/lst16_24.idc" METHOD=GET>
```

```
<SELECT NAME="Location">

<OPTION>WA

<OPTION>OR

</SELECT>

<INPUT TYPE=HIDDEN NAME="Id" VALUE="0">

<INPUT TYPE=SUBMIT>

</FORM>

</BODY>

</HTML>
```

Notice that we pass a value of zero the first time we call the idc. This makes the SQL statement return the first row that matches the location.

Select WA from the drop-down box and submit it to the idc. (See Figure 16.9.)

Figure 16.9. The first page of Washington companies.

Now click the anchor entitled Next 10 Records. The next set of records should show up, only this time there is only one left. (See Figure 16.10.)

Figure 16.10. The second page of Washington companies.

Go back to the selection screen and select OR; only three entries should show up. (See Figure 16.11.)

Figure 16.11. The only page of Oregon companies.

By changing the structure of the files a bit, we can let the user tell the idc how many files to retrieve at one time. Here is the htm that submits not only the location but also the number of files to display, using the parameter MaxRecords. (See Listing 16.24.)

Listing 16.24. HTML to calls lst16.28 with maximum number of rows.

```
<HTML>

<BODY>

<FORM ACTION="../scripts/lst16_28.idc" METHOD=GET>

Location: <SELECT NAME="Location">

<OPTION>WA

<OPTION>OR
```

```
</SELECT>

<BR>

Row to return:

5 <INPUT TYPE=RADIO NAME="MaxRecords" VALUE=5>

10 <INPUT CHECKED TYPE=RADIO NAME="MaxRecords" VALUE=10>

20 <INPUT TYPE=RADIO NAME="MaxRecords" VALUE=20>

<INPUT TYPE=HIDDEN NAME="Id" VALUE="0">

<INPUT TYPE=SUBMIT>

</FORM>

</BODY>

</HTML>
```

Notice that the radio buttons let the user select 5, 10, or 20 records. (See Listing 16.25.)

Listing 16.25. An Example of the MaxRecords attribute being used in an .idc file.

```
Datasource: WebSql

Username: sa

Template: lst16_29.htx

MaxRecords: %MaxRecords%

SQLStatement:

+SELECT Id, Name, Location

+FROM IDCCompany

+WHERE IDCCompany.Id >= %Id% AND IDCCompany.Location = '%LOCATION%'
```

In Listing 16.26, the MaxRecords input from the HTML form is inserted in the idc as MaxRecords.

Listing 16.26. The matching .htx file for Listing 16.25.

```
<HTML>
```

```
<BODY>

<TABLE BORDER=1>

<%begindetail%>

<%if CurrentRecord EQ 0%>

<TR>

<TD><B>Name</B></TD>

<TD><B>Location</B></TD>

</TR>

<%endif%>

<TR>

<TD><%Name%></TD>

<TD><%Location%></TD>

</TR>

<%enddetail%>

</TABLE>

<%if 'Id' EQ '0' %>

<%else%>

<A HREF=lst16_28.idc?

Id=<%Id%>&Location=<%Location%>&MaxRecords=<%idc.MaxRecords%>>

Next <%idc.MaxRecords%> Records</A>

<%endif%>

<%if CurrentRecord EQ 0%>

No Matches

<%endif%>

</BODY>
```

```
</HTML>
```

The only change to this file was to enable the user selection for the maximum number of records to be passed in the next records anchor. Each time the idc is called, it needs to know what the maximum number of records is. Notice that the htx uses a tag called <%idc.MaxRecords%> to do this. This tag is equal to MaxRecords that was passed to the idc, or in other words, the original number that was selected in the form. You may reference parameters that are passed into the idc from the form in the htx file by adding idc in front of them. For instance, in Listing 16.26 we could also reference <%idc.Location%>.

Summary

The IDC is a simple way to connect the IIS Web server to a database using ODBC. IDC lends itself to rapid application development. The files are text based, making them easy to change and maintain. The format isn't much different than HTML itself, making for a small learning curve. The biggest advantage is that it comes with the IIS; there is no additional cost for hooking the IIS to the database. Unfortunately, its simplicity leads to disadvantages. For all practical purposes, you can have only one query per Web page. Each Web page consists of two files that both have to be loaded from the hard drive for every hit, causing a decrease in performance.

Chapter 17

Microsoft ActiveVRML

by Garrett L. Clark

What Is ActiveVRML?

The name ActiveVRML refers to both a language and the ActiveX control that embodies that language. As the name implies, this technology was initially an outgrowth of virtual reality modeling language (VRML), a language for describing 3D scenes and objects. VRML gained instant acceptance within the Internet community when initially released in 1995, but the limitations of this specification were immediately obvious. Virtual worlds contained only static, nonmoving objects and provided no consistent way to handle user interactions. A surprising number of companies began to offer VRML "browsers," and each product had a different approach to the problems of interactivity and dynamic behaviors. There was clear consensus within the VRML community that a new standard was required to bring this great technology forward along its next evolutionary step.

Microsoft offered ActiveVRML as a successor to the VRML 1.0 specification and so joined the handful of organizations that had developed proposals for VRML 2.0. Although a competing specification was eventually chosen by the standards body, ActiveVRML remains a viable technology because of the unique strengths of the language and the versatility of Microsoft's implementation.

Rather than positioning the ActiveVRML control as a virtual-reality browser, Microsoft now sees it as an engine for active media of all types. By providing a rich, interactive language that can control images, sound, video, and 3D objects, ActiveVRML (AVRML) stands on its own as a premier Web-enhancement tool.

Language Specification

The language of ActiveVRML was first introduced to the public in December 1995 as part of Microsoft's initial offering of Web software. At a time when the future of VRML was being hotly debated, Microsoft clearly saw this specification as a contender to be the basis of VRML 2.0. Initial comments within the VRML community were extremely positive, and very few technical criticisms were ever leveled at the proposal.

Respect for the superiority of the language stems from both its completeness and the elegance with which it solves extremely complex behavioral problems. This well thought-out solution did not develop overnight by any means; rather, it is the product of many years of continuing research.

An important point to stress is that ActiveVRML is not a general-purpose language, such as C++ or Java. It is intended to address only one particular set of problems, but it does that very well. Because of this focus, AVRML is remarkably simple to learn and use compared to a language like Java.

True appreciation of this language, however, comes only from seeing it demonstrated live. Images, video, sound, text, and 3D shapes can all be blended and mixed in astonishing combinations. Start with a cube, and let it spin as it bounces around the screen. Then attach a sound that will spatially follow the cube, using stereo speakers to accurately trace its position. Give the cube some texture, but then go ahead and make one side a video screen for streaming MPEG data. Add 3D details to the cube to make it look like a TV set; then place it in a virtual living room. When the user clicks the nearby virtual remote, happily change video "channels." Then a beer floats in from the kitchen along a smooth spine trajectory, decelerating to stop in your virtual hand. There is no end to what could be imagined and created with these sorts of capabilities, but this is exactly what the language promises and what the ActiveVRML control delivers.

Microsoft's ActiveX Control

The current ActiveX Control implementation of ActiveVRML, released by Microsoft, is still the only AVRML product available, although the language is open for third-party exploitation. This control makes use of several other cutting-edge technologies such as Direct3D and URL Moniker support.

Caution

The last published version of ActiveVRML was still an Alpha release. This version will only run with the alpha version of Internet Explorer 3.0, which was released in April 1995. This ActiveVRML release was also not designed to work with the final release version of DirectX II, which incorporates the Direct3D rendering engine. Instead, use the prerelease version of Direct3D which is packaged with the ActiveVRML control.

Although the ActiveVRML control is still in Alpha release, it demonstrates much of the power of the language specification. But what really makes it stand out is the speed and responsiveness with which it displays animations of both text and images. Compared to the jerky, halting quality of existing Web-based presentation tools, such as Macromedia's Shockwave or Microsoft's PowerPoint viewer, the AVRML control is truly astounding in its performance capability.

Beyond its usefulness as a Web-enhancement tool, ActiveVRML represents the crowning achievement of Microsoft's component object model (COM) interface. The underlying advantage of AVRML is the language itself, but it is built on the solid foundations of several key technologies, all brought together with COM. Most notable are DirectDraw and Direct3D, which should serve as clear examples of just how

efficient this interface can be for performance-demanding applications. Communications are likewise built on new COM objects that handle asynchronous downloading of data.

The Maturing of VRML

The huge initial response to VRML has waned somewhat in the last year, largely due to the public's realization of the inherent limitations in this specification. People soon tire of wandering around virtual worlds where nothing moves, nothing reacts, everything is quiet, and there are no other people to interact with. Some of the proprietary solutions to these issues have been quite remarkable, but it remains to be seen whether VRML 2.0 will recapture the momentum initiated by its predecessor.

Ironically, the main limitation of VRML 1.0 will also be its enduring strength. As an accepted standard for describing static 3D objects, the specification forms a common point of interaction between all new modeling and animation software tools. The lack of behavioral programming is no obstacle when you are just seeking a platform-independent way to pass model data back and forth. The complex and time-consuming task of supporting VRML 2.0 is simply not required for this purpose.

ActiveVRML Versus VRML 2.0

Philosophically speaking, there is a great divide between the ActiveVRML language and the VRML 2.0 specification called "Moving Worlds, which grew from a joint proposal from Sun, Netscape, and others. One of the chief distinctions is VRML 2.0's forced reliance on external scripts using the Java language. This separation of model definitions from behavioral definitions adds tremendous complexity to the overall system, and consequently it is much more difficult for developers to implement applications based on this standard. Compared to the explosion of VRML browsers seen before and after the release of the 1.0 spec, only a few companies have announced products that support VRML 2.0.

ActiveVRML's complete integration of media definitions (including 3D objects) and behaviors is the cornerstone of its elegance. This direct approach has the added benefit of simplifying the task of writing authoring tools. Microsoft has done its part to provide a control for displaying any given AVRML file, but it will fall to third-party software vendors to create great tools for producing these files in the first place. Translating a user's vision into actual lines of code can be very difficult with a general-purpose language like Java, but it is an extremely natural process with AVRML.

It's just this sort of strength that will allow ActiveVRML to persist, though it was seemingly defeated by VRML 2.0. Few people will ever take up the challenge of learning Java just to make a cube spin around in space. Average people will be able to create active, interactive, 3D worlds only when the tools and the process are intuitive enough. Those tools can be more readily developed with AVRML.

Beyond the issues of dynamic 3D worlds, ActiveVRML possesses other key capabilities that put it in a class of its own. The seamless way it works with all different types of media, not just 3D objects, is remarkable. Also notable is the time-based nature of all behaviors, which assures an author that viewers

will see activity in a similar way regardless of the processing power found in the viewer's computer. These distinctions from VRML 2.0 serve to show the holes still left in that specification.

Using the ActiveVRML Control

Installing and using ActiveX controls is supposed to be the simplest task a user could ever face. In most cases, a user might never be aware that a program had been downloaded and was running within a Web page. Unfortunately, the ActiveVRML control falls a little short of that goal. Most users of Internet Explorer 3.0 will have to pay attention to a few important issues before they can begin animating their sites.

Installation Issues

Unlike most other ActiveX controls, ActiveVRML is not designed to be installed automatically using the Component Download API. A more elaborate installation is required because of the product's reliance on DirectX. At some future point, DirectX will probably ship as part of the Windows operating system, allowing controls like AVRML to be more easily downloaded. But in the meantime, the installation program must be downloaded (or loaded from the CD) and manually run by the user.

Another thing to be aware of is that the current Alpha release of ActiveVRML does not install the control's OCX file in the \System\Occache directory, where controls normally reside; instead, it places the required files into a separate folder within the Internet Explorer directory. Little quirks like this are understandable when you realize this program was among the very first ActiveX controls ever released, long before the current specifications had been finalized. Certainly, the next version of AVRML will take full advantage of the finalized ActiveX standards.

The 3D rendering capabilities of ActiveVRML are built on top of the new Direct3D engine that is a part of DirectX II and DirectX 3. DirectX is a set of technologies designed to maximize the performance of games and other real-time applications, under both Windows 95 and Windows NT 4.0. The DirectX API gives programmers powerful access directly to the video and sound cards, and it takes full advantage of any special hardware capabilities found on those devices. In the case of Direct3D, that means computer systems with advanced 3D accelerated graphics cards will see a dramatic boost in speed and/or image quality. This new breed of graphics card is becoming more and more common these days, but for those computers with ordinary graphics, Direct3D still turns out a top-notch performance.

Caution

The current (alpha) release of the ActiveVRML control is not compatible with the final version of DirectX II or with DirectX 3. The AVRML setup program will automatically install a compatible beta version of DirectX II, but any newer versions must first be removed.

Invoking from HTML

Like other ActiveX controls, ActiveVRML can be embedded into an HTML document by using the OBJECT tag. The control has its very own globally unique identifier (GUID), which serves to tell your browser precisely what script you want loaded. Here is a sample use of the OBJECT tag that might be used to invoke the testcode.avr script:

```
<OBJECT
    CLASSID="clsid:{389C2960-3640-11CF-9294-00AA00B8A733}"
    ID="AVView"
    WIDTH=256 HEIGHT=256>
    <PARAM NAME="DataPath" VALUE="testcode.avr">
    <PARAM NAME="Expression" VALUE="model">
    <PARAM NAME="Border" VALUE=FALSE>
</OBJECT>
```

If you have ever incorporated ActiveX controls into HTML before, two things will be noticeable immediately about this sample. The first is the outdated syntax of the CLASSID parameter. The outdated syntax would be

```
CLASSID="clsid:{389C2960-3640-11CF-9294-00AA00B8A733}"
```

More recent code would use this syntax:

```
CLASSID="clsid:389C2960-3640-11CF-9294-00AA00B8A733"
```

TIP

The exact syntax of the OBJECT tag has been something of a moving target as

the development of Internet Explorer 3.0 edged towards completion. The specification for this tag is not under the control of Microsoft; it is the responsibility of the World Wide Web Consortium (W3C) which acts as an impartial standards organization for many Web-based technologies (see http://www.w3.org)

This is not a big deal, but you should keep it in mind if you are writing HTML for both ActiveVRML and other ActiveX controls. Also, be aware that when the next release of ActiveVRML comes out, your HTML source code will have to be revised to use the newer syntax.

Astute readers will note that there is no CODEBASE parameter. Normally, this information would tell the browser where to find a copy of the control for downloading, if it was not already installed in the system. Because this control does not yet support the Component Download model for automatic installation, using CODEBASE would be meaningless. The next release of AVRML will conform fully with the new specifications.

The WIDTH and HEIGHT parameters are used just as with any control to specify the size of the screen area used. With ActiveVRML, however, there is an additional consideration when planning for the size of your active display. DirectDraw and Direct3D each require significant portions of video memory to store special-purpose buffers. The larger the desired display, the more video memory is required. Most graphics cards come with 2M of memory these days; many carry 4M, and a very few have 8M. But a tremendous number of cards are still being used that only have 1M of video memory—typically on older 486 systems. Web-site builders have to be aware of who their target audience is and what the minimum hardware requirements are. In general, a 256*256 display will work for most of the PCs on the Internet today. Predicting exact memory usage is difficult because of the wide variety of features and behaviors found on each of the cards available these days, and it further depends on the design of the video driver, the user's current display resolution, and the color depth being used.

In addition to the standard parameters supported by the OBJECT tag itself, ActiveVRML accepts several other custom values passed through the PARAM tag. The names of these parameters match the exposed COM interfaces supported by AVRML, and they are used for setting up the initial state of the control. The foremost of these is DataPath, which specifies the URL of the ActiveVRML script that needs to be loaded and executed. Note that this script is not loaded by the browser but by the control itself using URL Moniker objects.

The Expression parameter refers to an expression found within the specified ActiveVRML file (AVR file). If this expression does not have a match in the target script, an error will be generated and nothing will be displayed. This AVRML interface can be dynamically controlled to change which expression is currently evaluated, for files that may contain multiple expressions.

The meaning of the Border parameter is pretty self-explanatory. You can enable a thin border to frame the AVRML control, or you can leave it off. It's absolutely up to you.

Interacting with JavaScript and Visual Basic Script

Even though the ActiveVRML language has many advantages over Java, BASIC, and even C++ for the purposes of describing active media content, there are times when these general-purposes languages can come in handy. Tasks like database integration, coordinating multiuser environments, or communicating with other controls all require functionality that is not found in AVRML. However, the flexibility of AVRML's COM interface provides easy integration with any language that fits Microsoft's scripting object model. Currently that means Visual Basic Script and JavaScript. Through such an intermediary script, AVRML can also interact with other ActiveX controls and even the browser itself.

Script interactions with the ActiveVRML control flow in two directions—into the control through both exposed properties and externally fired events and from the control through internally generated events. Custom-defined events provide an extremely flexible way to tie together an external script and an AVR file.

The ActiveVRML control has properties that expose a fixed set of functionality. The following list names the exposed methods that can be manipulated by external scripts or other COM-aware programs, along with the data type they each accept.

- GetDataPath (BSTR)
- SetDataPath (BSTR)
- GetExpression (BSTR)
- SetExpression (BSTR)
- GetBorder (BOOL)
- SetBorder (BOOL)
- GetFrozen (BOOL)
- SetFrozen (BOOL)
- FireImportedEvent (I2, VARIANT)
- AboutBox ()

Communications from ActiveVRML

The DataPath, Expression, and Border methods correspond to the PARAM tags used to initialize ActiveVRML. It is possible to change these values dynamically, but this action requires use of the SetFrozen method. When the Frozen property is TRUE, evaluation of the current expression is halted. After the changes to DataPath and/or Expression have been made, the control can be unfrozen again.

Here is a sample of HTML that uses Visual Basic Script to choose between two different AVR files for viewing:

```
<HTML>

<HEAD>
```

```
<TITLE>ActiveVRML Test Page</TITLE>
</HEAD>
<BODY>
<OBJECT
    ID="AVRCtrl"
    CLASSID="clsid:{389C2960-3640-11CF-9294-00AA00B8A733}"
    WIDTH=256 HEIGHT=256>
    <PARAM NAME="DataPath" VALUE="File1.avr">
    <PARAM NAME="Expression" VALUE="model">
    <PARAM NAME="Border" VALUE=TRUE>
</OBJECT>
<BR>
<INPUT NAME="File1" TYPE=Button VALUE="View File #1">
<INPUT NAME="File2" TYPE=Button VALUE="View File #2">
<SCRIPT LANGUAGE="VBScript"><!--
    sub File1_onClick
        AVRCtrl.Frozen = TRUE
        AVRCtrl.DataPath = "File1.avr"
        AVRCtrl.Frozen = FALSE
    end sub
    sub File2_onClick
        AVRCtrl.Frozen = TRUE
        AVRCtrl.DataPath = "File2.avr"
```

```
        AVRCtrl.Frozen = FALSE

    end sub

--></SCRIPT>

</BODY>

</HTML>
```

If you prefer to use JavaScript, the SCRIPT tag would only have to be slightly modified, and the rest of the HTML would be unchanged. The JavaScript version would look something like this:

```
<SCRIPT LANGUAGE="JavaScript"><!--

    function File1_onClick ()

        {

        AVRCtrl.Frozen = TRUE;

        AVRCtrl.DataPath = "File1.avr";

        AVRCtrl.Frozen = FALSE;

        }

    function File2_onClick ()

        {

        AVRCtrl.Frozen = TRUE;

        AVRCtrl.DataPath = "File2.avr";

        AVRCtrl.Frozen = FALSE;

        }

--></SCRIPT>
```

This sample made several assumptions for simplicity. Because the script does not specify a new Expression, the one specified within the OBJECT tag is still in force. Both File1.avr and File2.avr must then implement that expression within the AVR code. You will learn a lot more about how to do that a

little later on. Also, by not specifying a full URL to the AVR files, the programmer is assuming that they reside in the same directory as the HTML document itself.

The method that has the most interesting possibilities by far is FireImportedEvent, which allows a script to trigger a custom event within the running AVR code. Each event to be used will have its own unique identifier, defined in both the external script and within the AVR file. Optional data may be passed along with an event and can be either a string or a double floating-point value. Here is an example of how to fire a custom event from Visual Basic Script:

```
<SCRIPT LANGUAGE="VBScript"><!--

    sub File1_onClick

        AVRCtrl.FireImportedEvent (1, "Event Number One")

    end sub

--></SCRIPT>
```

Communications to ActiveVRML

Sending commands to ActiveVRML is important, but equally so is the ability to receive information back. This functionality is implemented through AVRML's ActiveVRMLEvent interface, which is triggered internally and meant to be handled by an external script. As with external triggers, data may be passed along, this time coming from AVRML instead of to it. Visual Basic Script would typically handle this event with the following syntax:

```
<SCRIPT FOR="AVRCtrl" EVENT="ActiveVRMLEvent(EventID, Param)"

    LANGUAGE="VBScript"><!--

Select Case EventID

    Case 1

        Text1.Text = Param

    Case 2

        Text2.Text = Param
```

```
    End Select

--></SCRIPT>
```

The Param value can be either a string or a double float, or it may consist of nothing. EventID is an identifying integer.

The JavaScript version of the same code is:

```
<SCRIPT FOR="AVRCtrl" EVENT="ActiveVRMLEvent(EventID, Param)"
    LANGUAGE="JavaScript"><!--
    switch (EventID)
        {
        case 1:
            Text1.Text = Param;
            break;
        case 2:
            Text2.Text = Param;
            break;
        }
--></SCRIPT>
```

Displaying Your Own Active Content

So far, this chapter has looked at how to use ActiveVRML without talking much about why you're using it. Creative Web designers already have lots of experience using the traditional media types found on the Internet, and AVRML builds on that foundation of existing content with the added dimension of dynamic activity.

Available Media Types

The ActiveVRML language relies heavily on the capability of importing content in a wide variety of data formats. As shown in Table 17.1 images, sounds, and 3D objects are brought into the AVRML scene where you can manipulate them at will.

Table 17.1 Supported media types.

3D Formats Image Formats Video Formats Sound Formats VRML (*.wrl) JPEG (*.jpg) AVI (*.avi) WAVE (*.wav) DIRECTX (*.x) GIF (*.gif) MPEG (*.mpg) AU (*.au) BMP (*.bmp) QUICKTIME (*.mov) AIFF (*.aiff) MIDI (*.mid)

Images and video are internally rendered onto DirectDraw surfaces, which can then be used as textures for 3D objects. Surfaces can also be manipulated and combined with other surfaces to create stunning special effects. Within an AVR script, this is how to import an image:

```
myimage = import ("picture.jpg");
```

The ActiveVRML function import accepts the URL of a media resource and returns a value used to define myimage. This value has a type that depends on the media format being imported. For instance, imported GIF files produce the type image, and WAV files generate the type sound. The concept of types is very important to AVRML and is explored in the next sections of this chapter. The resulting definition, myimage, can now be used in any expression that accepts type image.

You can do some powerful things with image types, such as performing cropping, setting opacity, and tiling the image over an area. But the most compelling functionality is the ability to apply 2D transformations such as scaling, rotating, and translating. Individually composited images can also be grouped into a montage type. A montage maintains a depth value for each image in the collection and can combine the various layers to form a final rendering value (of type image, of course). With just these capabilities and nothing else, ActiveVRML would be a really useful tool for laying out Web pages. And you haven't even gotten near the good parts yet, so hold on.

Once you've played with the spatially oriented sounds in ActiveVRML, boring 2D sound will never again be enough. By combining a sound type with a geometry type, which carries with it three-dimensional positioning information, a sound is reproduced in 3D space. The effect requires stereo speakers, of course. Consider the example of a fighter plane model crossing the screen from left to right with jet engines roaring. The sound will track the rendered image as it moves, and then attenuate as the virtual plane moves into the distance. Even the Doppler effect from swiftly moving sound sources is approximated. The truly wonderful aspect of this technology is that it is all done for you; ActiveVRML takes care of all the hard work after you've made a few simple definitions.

You can manipulate sounds in other ways as well. The looping and mixing functions of ActiveVRML make it easy to produce the complex sound behaviors needed for immersive simulations. Continuous ambient tracks are seamlessly mixed with event-driven sounds, and the Web designer does not have to

mess with the details. One of the most extraordinary features of sound types is their capability of dynamically adjusting playback rates. This is a great way to transform a deep voice into a chipmunk, or the reverse.

3D Objects

Even though ActiveVRML was once positioned as the successor to VRML, AVRML's support for 3D primitives is limited to importing existing VRML models (*.wrl files). Imported files produce values of type geometry and are defined as in this sample:

```
(mymodel, minExtent, maxExtent) = import ("testmodel.wrl");
```

Two additional definitions, minExtent and maxExtent, are of type point3 and contain information about the bounding box of the imported geometry. It must be noted that mymodel now refers to the entire VRML file, regardless of how many individual objects are contained within that file. This is a serious drawback if you want to manipulate specific objects, but you can overcome it with a little bit of good planning when you design the VRML content.

VRML worlds are formed from a collection of components, called nodes, which are arranged into a hierarchical structure known as a scene-graph. Some nodes in the scene-graph refer to geometrical primitives, and others describe textures, lights, cameras, and transformations. Each node possesses properties that have been passed down from upper layers of the hierarchy, and in turn, it applies those properties to the nodes below it. A given texture node, for example, may be applied to several cubes, whereas some nearby spheres follow a different texture node. A complex world may contain hundreds or thousands of nodes, each of which represents some aspect of the virtual scene. Unfortunately, ActiveVRML treats all the nodes as a single unit and cannot control the individual pieces.

Not only does this prevent you from treating nodes as individual entities, but if your AVRML script calls for a texture to be applied to an imported model, it will be applied to all geometries found in that file's scene graph. You may have wanted to wallpaper the virtual living room, but you end up papering the TV and sofa too.

Although ActiveVRML accurately renders imported VRML 1.0 worlds, all of the original properties of the VRML geometries are superseded by any properties applied by the ActiveVRML script. AVRML textures take precedence over VRML textures, for instance.

The way to handle all this is pretty simple, as long as you don't have to work with a lot of pre-existing scenes. Even then, your modeling tools should be able to break up a complex scene into individual components. The idea is to keep the VRML files very simple, one geometry per file, and then import all the required 3D objects as true individual entities. It does not matter if textures and colors are specified in the VRML or AVRML side, but a consistent approach would keep designs much simpler. In general, it is probably best not to use any VRML functionality if it can be duplicated with AVRML.

Once you have imported a model, you can aggregate it with other 3D objects to form groups that can be collectively manipulated. These other objects could be imported shapes but could also be lights or sounds.

A quick word about aggregation: many basic types, such as geometry, have the ability to be joined with objects of similar type. The way that the resulting aggregate is formed depends on the types involved. Combined geometry objects are the geometric union of all enclosed shapes and lights, whereas combined sound objects mix their individual tracks to form a single output. Values of type image are combined by joining one image over the other. The ability to create complex compositions from simple components is one of the strengths of the ActiveVRML language, as you will see in the next section.

Writing an ActiveVRML Script

By now, you are probably eager to see ActiveVRML at work and to try out some of your own AVRML scripting. The following sample file will walk you through the complete steps required and will highlight some of the other key elements of the language.

Here is the enclosing HTML framework (see Listing 17.1) that will invoke this sample. It is kept as simple as possible, to allow focusing on the actual AVRML script.

Listing 17.1. Saucer1.html.

```
<HTML>

<HEAD><TITLE>Saucer1 Test Page</TITLE></HEAD>

<BODY BGCOLOR=WHITE><CENTER>

<OBJECT

    ID="AVRCtrl"

    CLASSID="clsid:{389C2960-3640-11CF-9294-00AA00B8A733}"

    WIDTH=300 HEIGHT=150>

    <PARAM NAME="DataPath" VALUE="sample1.avr">

    <PARAM NAME="Expression" VALUE="model">

    <PARAM NAME="Border" VALUE=TRUE>

</OBJECT>
```

```
</CENTER></BODY>

</HTML>
```

As you can see, it really doesn't take much code to implement the AVRML control, unless you choose to interface with HTML in a more dynamic way, which you will do later on.

First Script

This first script is designed to simulate something most of us have encountered at some time or another: low-level flying saucers. Our virtual UFO will skim along the mountain tops, emitting eerily authentic sound effects. When the script is run, the results are as shown in Figure 17.1.

Figure 17.1. The results of Saucer1.avr.

Make sure that this script (see Listing 17.2) is located in the same directory as the enclosing HTML file. If it is not, the URL specified in the HTML file must be changed to reflect the actual location of the AVR file.

Listing 17.2. Saucer1.avr.

```
1: // ActiveVRML 1.0 ASCII

 2: // Saucer1 Sample Script

 3:

 4: // The first step is to import the raw media for our sample

 5: ship = first (import ("shipred.gif"));

 6: mountain = first (import("mountain.gif"));

 7: clouds = first (import("clouds.gif"));

 8:

 9: // Define a 2D transformation to describe the saucer's position

10: movement = translate (0.005, 0.007);

11:
```

```
12: // Define a new saucer object that has movement applied to it

13: saucer = transformImage (movement, ship);

14:

15: // Define the expression used for display output

16: model = mountain over saucer over clouds;
```

NOTE

The line numbers in this script are for reference purposes only, and should not be typed in any actual script.

This initial script demonstrates some of the essential elements of an ActiveVRML program. Line 1 is a required comment that identifies this file as containing an AVRML 1.0 program, formatted with plain ASCII. This is the only kind of file that the control will accept.

Lines 5, 6, and 7 define three objects formed from imported graphics files. These objects are implicitly declared as being of type image. The import function actually returns three values, but the first function filters out all values but the image value. Later on, this example will show how to use the additional values, which specify the image's size and resolution.

In Line 10, another data type is defined, called a transform2. This transformation type is applied to two-dimensional points, vectors, and images to specify how they are positioned in the plane of the screen. The declared object, movement, is assumed to be a transform2 because that is the return type of the translate function when called with two number parameters. Later uses of this definition must be within a context that also implies it is of type transform2, or type mismatch errors will occur.

The previously defined transformation is now applied to the desired image using the transformImage function, as seen in Line 13. Actually, it is more correct to say that an entirely new object is declared that has all the same properties as the source image but has been modified by the transform2 object. In this case, the values used to set up this transformation are a couple of hard-coded numbers that place the saucer above and to the right of the screen's center.

Line 16, the final line of the script, provides an expression for the definition model. This definition is the same one specified in the HTML file, within the Expression parameter of the OBJECT tag. The value of this definition is evaluated to produce the current display. There could be several such definitions that provide top-level directives, but only one is active at any given time. In the case of this sample, the expression is evaluated to be an image composed of three successive layers built using the image type's over function.

Use of the over function introduces the concept of *composition*, one of the primary features of the ActiveVRML language. Composition enables complex objects and behaviors to be assembled from simpler definitions.

At this point, you are ready to run the control to see the script in action, which you can do by opening the

Saucer.html file. Actually, the term *action* is used loosely, because it doesn't really do anything active at this point. You'll fix that next.

If you introduced any mistakes into the script as you prepared it for the ActiveVRML control, by now you have seen the error-handling capabilities the control possesses. A dialog is displayed to indicate both the row and column numbers where the error occurred. In a case of multiple errors in your script, however, only the first error is actually reported.

Enhancing the Script

So far, this is easy, but that's because nothing is happening yet. Listing 17.3 shows how a few extensions to the previous script can add both rudimentary motion and some dynamic appearance changes. The flying saucer will now travel from the left side of the screen across to the right, while alternately flashing a red and blue light. For the sake of simplicity in this sample, nothing special happens when the saucer goes past the right side of the screen (and off into the void, maybe?). Notice the cool effect of having the saucer glide *behind* the mountain peaks, thanks to the use of layering in the design.

Listing 17.3. Saucer2.avr.

```
1: // ActiveVRML 1.0 ASCII
 2: // Saucer2 Sample Script
 3:
 4: // The first step is to import the raw media for our sample
 5: shipred = first (import ("shipred.gif"));
 6: shipblue = first (import ("shipblue.gif"));
 7: mountain = first (import("mountain.gif"));
 8: clouds = first (import("clouds.gif"));
 9:
10: // The saucer will alternate between two different images
11: saucer = shipred until predicate (time > 1) =>
12:     (shipblue until predicate (time > 1) => saucer);
```

```
13:

14: // Define a 2D transformation to describe the saucer's position

15: movement = translate (0.005 * time - 0.05, 0.007);

16:

17: // Define a new saucer object that has movement applied to it

18: activesaucer = transformImage (movement, saucer);

19:

20: // Define the expression used for display output

21: model = mountain over activesaucer over clouds;
```

As you can see in Lines 5 and 6, there are now two versions of the saucer image defined. One has a red light on top and the other has blue. The definition that spans Lines 11 and 12 sets up a simple behavior that causes the displayed saucer to alternate between the two images once per second.

Having a periodic behavior implies that time can be examined in ActiveVRML. In fact, time is one of the most important concepts in AVRML because *all* values are time-varying. Not just numbers, but images, geometries, and other types can all be considered forms of behaviors because they have the capacity to change with time.

Temporal manipulations rely on the implicit time property associated with every behavior. The value of time is of type number, and it represents the behavior's local elapsed time in seconds, starting at 0. This is where it starts to get tricky, so read carefully. A behavior's time is initialized whenever that behavior's definition changes. In the example, saucer is a behavior with a value that alternates between shipred and shipblue, resetting time after each switch.

TIP

Always keep in mind that any change to an object's behavior will result in its time restarting at zero.

In this sample, you want the behavior of saucer to change once each second. ActiveVRML provides a way to signal a behavior by triggering an event that is specified in the behavior's definition. Events are a very flexible way to tie together all sorts of reactive behaviors, and several functions are available for manipulating this type. In this instance, the predicate function is used to trigger an event when the specified condition is TRUE, which is when the value of saucer's time is greater than one. This event satisfies the until condition, and a new saucer is created with the definition on the right side of the =>. Here is another example for clarity:

```
texture = bumpy until (some event) => smooth;
```

The event does not have to be declared in the same definition as the reacting behavior. Events can have their own defined names, like other data types. These two lines are equivalent to the preceding snippet:

```
myevent = (some event);

texture = bumpy until myevent => smooth;
```

Now you can see just what is happening in Lines 10 and 11. When the script first begins to run, saucer is initially defined as shipred, and time begins counting from zero. After one second has elapsed, predicate generates an event that changes the definition of saucer to an almost identical definition based on shipblue. The new definition also specifies a reaction to a time-driven event, and after another second elapses, saucer is redefined yet again—this time as itself. What this really means is that you start all over again with the original saucer definition (at time zero, of course). This ability to have self-referential definitions is essential for building recursive functions, as you'll see later on.

The transformation that is defined in Line 15 is also based on time, but in a much more direct way. The value of movement is time-varying, beginning at zero and ever-increasing. Nothing in this script will ever change that behavior, so the value of time is never reset. It is important to note that movement's time is not the same as saucer's time—every object in ActiveVRML maintains its own relative measurement of time, starting at the moment it was instantiated.

The rest of the script is functionally the same as that in the first version. The value of the output definition, model, is still evaluated as the composition of three separate images regardless of whether the images are static or changing. To stress an important point once again, in ActiveVRML *all* values are potentially time-varying.

Final Touches to Script

With some of the basic concepts now covered, you can finish off the script with a few dramatic flares. When the flying saucer drifts off the right side of the background image, it will now reappear on the left side. There is also a sinusoidal vertical component to the saucer's wandering flight path. And finally, the sound effects are added to top off the realism of this simulation.

Listing 17.4. Saucer3.avr.

```
1: // ActiveVRML 1.0 ASCII

 2: // Saucer3 Sample Script
```

```
 3:
 4: // The first step is to import the raw media for our sample
 5: shipred = first (import ("shipred.gif"));
 6: shipblue = first (import ("shipblue.gif"));
 7: mountain, extents = import ("mountain.gif");
 8: clouds = first (import ("clouds.gif"));
 9: shipnoise = loop (first (import ("ship.wav")));
10:
11: // The saucer will alternate between two different images
12: saucer = shipred until predicate (time > 1) =>
13:     (shipblue until predicate (time > 1) => saucer);
14:
15: // Identify the components of the image boundary
16: maxx = xComponent (extents);
17: maxy = yComponent (extents);
18:
19: // Make speed a function of the image size
20: velocityx = maxx / 5;
21: velocityy = velocityx;
22:
23: // Modify the x and y positions
24: xpos = (velocityx * time - maxx) until edgeevent => xpos;
25: ypos = sin (time) * velocityy + (maxy / 4);
```

```
26:
27: // Define an event to handle moving off the right side of the image
28: edgeevent = predicate (xpos > maxx);
29:
30: // Define a 2D transformation to describe the saucer's position
31: movement = translate (xpos, ypos);
32:
33: // Define a new saucer object that has movement applied to it
34: activesaucer = transformImage (movement, saucer);
35:
36: // Add some effects to our sound
37: soundgain = ypos / maxy  + 0.2;
38: soundeffect = gain (soundgain, rate (2, shipnoise));
39:
40: // Define the expression used for display output
41: model = mountain over activesaucer over clouds, soundeffect;
```

The first thing that's noticeably different is the way the mountain image is being imported (Line 3). Instead of only accepting the image type, you are also acquiring the image's size information. By default, an untransformed image is centered at coordinates (0, 0), so the position of any corner is sufficient to compute the overall boundary dimensions. This data is returned in a vector2 type, which can be formed from any two numeric values. This information will be used later to determine when the saucer is moving off the edge of the background picture.

In Line 9, a sound type is imported and defined so that the playback loops continuously. The import function of sound types actually returns values for both the left and right channels, and for the duration of the longest channel. If the imported sound is monophonic, the same value is given to both channels. This script does not need to process individual sound channels, so the first function is used again to restrict the return values.

In the next several lines, the saucer's behavior is constructed from a few simple components. Definitions in Lines 16 and 17 extract individual x and y boundary values (as type number) from the original vector2.

Then some velocity values are defined that depend on the size of the boundary image. The horizontal and vertical velocities are maintained separately, with chosen values that result in a 10-second horizontal traversal of the screen.

Line 24 is the actual definition for the horizontal position of the saucer. When time is zero, this value places the saucer at the left side of the screen. After the ship passes over the right side, edgeevent is triggered to change the behavior of xpos. The new definition is recursive, which in this case simply serves to reset time to zero.

Vertical position follows a sin wave, as calculated in Line 25. ActiveVRML has a full range of mathematical functions, such as sin, which you can use to construct very complex behaviors.

Next, edgeevent is defined, which is used to possibly modify the value of the horizontal position (see Line 24). This event is defined as a simple predicate condition that is triggered when the saucer's position falls across the right boundary.

The horizontal and vertical positions are used in Line 31 to declare a transform2 object. This is similar to the previous sample, but in this case both parameters to the translate function are time-varying behaviors. This transform2 is applied to the static image of the flying saucer, resulting in an image appropriately named activesaucer (in Line 34).

You could have stopped right there and had a pretty good simulation, but it turns out that the audio sample doesn't actually sound much like an alien spaceship. No problem! In Lines 37 and 38, you not only speed up the playback rate (by a factor of 2), but you also tie the playback volume to the saucer's altitude on screen.

The last definition shows how a sound channel is simply added as another parameter to the output expression. If there were two sound channels, they would both be added as parameters.

Now that you've seen some brief examples of how ActiveVRML is structured, the rest of the chapter can go into some detail about the specifics of the language.

ActiveVRML Language

With some idea now of how ActiveVRML scripts are actually written and implemented, learning some of the more specific details will be easy. This section will focus on the elements of the language and the different ways those elements can be assembled.

Note

This section is not intended as an exhaustive reference, but as a synopsis of the general functionality found in the ActiveVRML language. The final release of the ActiveVRML control will include a specification containing complete technical details.

Language Overview

An ActiveVRML program is simply a collection of declarations that define objects of various types. These named objects have values that vary with time and in reaction to event-driven interactions with other objects. Declarations are composed of identifiers and expressions that are evaluated with strict attention to object typing. A single declaration must be present that has also been named in the container HTML file; it is the one evaluated to produce any actual output.

Data Types

There are several fundamental types used in ActiveVRML, and all identifiers and expressions must correctly evaluate to one of these types when the script is compiled. In most cases, the actual type does not have to be specified in the code because AVRML can infer it from the context in which the identifier or expression is used.

Here is how typing is applied to a simple declaration:

```
bird = import ("bird.gif");
```

The identifier bird is of type image. The function import is of type image * vector2 * number. You could have extracted the additional returned information like this:

```
(bird, birdextent, birdresolution) = import ("bird.gif");
```

It will be assumed that birdextent is of type vector2, and that birdresolution is of type number, when these identifiers are used elsewhere in the script. Incorrect usage always generates a type mismatch error from the AVRML control.

The type associated with a function definition will indicate its return value as well as the input parameters. Take a look at this example:

```
double = x * 2;

result = double (20);
```

The function product is of type number-> number. This notation indicates that the function accepts a

number as a parameter and yields an object of type number.

If the preceding function was a little more general, it could demonstrate the polymorphic properties of types:

```
product (x, y) = x * y;
```

The function type is now (a, a) -> a, where a indicates a generic type identifier, so the parameters and return value can be of any type that supports the * operator, for example:

```
result = product (20, 2);

scaledvector = product (2, vector2Xy (20, 20));
```

The first case demonstrates a usage of number * number-> number and the second case shows an alternate usage: number * vector2 -> vector2.

You can directly specify an identifier's type when it is declared, by following the name with : type as in this definition:

```
product (x : number, y : number) = x * y;
```

Sometimes the ActiveVRML control is not able to compile your script when it encounters type ambiguities that it cannot resolve. This is not necessarily an error in your code, but it will force you to explicitly type the offending definition. In general though, AVRML is very good at determining types just from context alone. Whether or not your objects are explicitly typed, as a designer you must always be aware of object types so you can use them correctly throughout the script.

Image Type

The image type is of primary importance to ActiveVRML scripts and is the ultimate type to which the output expression must evaluate. An image does not represent a static picture but is instead a behavior that can vary over time, for instance:

```
plainimage = import ("parrot.gif");

opaqueimage = opacity (1 / time, plainimage);
```

The import function, which you've seen many times now, is a constructor function of the image type. opacity is another image function, of type number * image -> image, which is used to construct a new object that has a specified opacity value between 0.0 and 1.0 (0.0 is completely transparent). In this case, the resulting image will begin to fade away immediately, as its opacity drops towards zero with the passing of local time.

There are two other methods to create image objects, using 3D shapes and text. A geometry object is used in conjunction with a camera object to produce an image using the function renderedImage (type geometry * camera -> image. See the next section for more details about this process. An image can also be produced using the renderedText function (text -> image * point2), which returns both an image and the point2 coordinate of the upper right corner of the resulting text.

Combining images, as with a foreground and background, is extremely simple with the over function (type image * image -> image), which combines any two source images to produce a new third image object. All images are centered at (0, 0) by default, so overlapped images should first be translated into position. If you had two images, each 100*100 pixels in size, and you wanted to place them on the screen side by side, you could use the following code:

```
parrot = import ("parrot.gif");

newparrot = transformImage (translate (-50,0), parrot);

raven = import ("raven.jpg");

newraven = transformImage (translate (50,0), raven);

birds = newparrot over newraven;
```

The function transformImage is of type transform2 * image -> image, and so it requires a transform2 object. In this case, translate was used to produce an appropriate object, but many other transformation functions are available and are discussed in a following section. If the translation was changed so that the images overlapped slightly, newraven would be obscured by newparrot. On the other hand, if newparrot had been defined with some level of opacity, then the images could actually be blended.

The remaining two image functions are crop and tile, each of type point2 * point2 * image -> image. In both functions, the two point2 parameters define a rectangular patch of the source image by specifying the area's minimum and maximum extents. In the case of crop, the supplied image is cropped to the indicated size. The tile function also crops the original image to the indicated size, but then it also replicates the image across the entire new surface, which has infinite extent.

Geometry Type

The geometry type is what puts the "VRML" in ActiveVRML. Geometries are behaviors, like image

types, in the sense that their values are time-varying and reactive. The 3D nature of geometry objects allows them to be manipulated by a rich set of transformations, but it does not specify how the results are to be viewed on a 2D display screen. That is controlled by the image function renderedGeometry (see the Image Type section).

The current version of AVRML has only one way to create visible geometry objects, which is to import the data from VRML 1.0 files. Although the internal representation of this type is comprised of vertex and triangle definitions, there is no way to manipulate these individual data points; they can only be modified as a whole. Here is how data can be loaded and displayed:

```
parrot3d = import ("parrot.wrl");

parrotimage = renderedGeometry (parrot3d, defaultCamera);
```

ActiveVRML uses a Cartesian right-handed coordinate system to describe the three-dimensional virtual environment. This means that the positive X axis points to the right of the screen, the positive Y axis points up, and the Z axis grows out of the screen towards the viewer. When you are looking along an object's axis of rotation in its positive direction, the object appears to turn clockwise when positive rotation is applied.

Imported geometries are combined into larger aggregates using the union function, of type geometry * geometry -> geometry. The resulting definition can be used to collectively manage a group of objects, each of which is still individually defined.

The most commonly used function of a geometry type is transformGeometry (type transform3 * geometry -> geometry), which applies a given transform3 object to a specified geometry. This function works in a very similar way to the image function transformImage, but here the functionality is extended into a three-dimensional environment.

Also similar to the image type is the geometry function opacity3, which specifies the degree of transparency applied to a visible geometry. It is of type number * geometry -> geometry.

The material appearance of a rendered geometry object is determined by the combination of light and the shading properties of the object. These properties are controlled with five related functions:

- ambientColor(type color * geometry -> geometry) determines how a geometry's material will reflect ambient light within a scene, with the default being white.
- diffuseColor (type color * geometry -> geometry) specifies how a material reflects light from any defined lights that it is aggregated with. The default diffuse color is white.
- emissiveColor(type color * geometry -> geometry) is color that emits from a material without regard for external illumination properties. Typically, emissive color is used to define pre-lit vertices in an object so that the overhead of lighting calculations is avoided. The default is black.
- specularColor(type color * geometry -> geometry) determines the color of a material's specular highlights, which controls the apparent shininess. The default is black, which results in a flat, non-shiny surface.
- specularExponent(type number * geometry -> geometry) controls the degree of shininess of an object's surface material. The default exponent value is 1.0.

One of the most powerful features of ActiveVRML's 3D rendering is the ability to use texture maps on geometry objects. Texture maps are built with image objects and are applied to a given shape using texture coordinates specified in the original VRML file. AVRML does not calculate any default texture coordinates if none are available, so the texture is simply not drawn. Texture mapping is applied using the texture function, of type image * geometry -> geometry. Here is an example usage:

```
feathers = import ("feathers.gif");

parrot3d = import ("parrot.wrl");

featheredparrot = texture (feathers, parrot3d);

parrotimage = renderedGeometry (featheredparrot, deafultCamera);
```

Because image objects are essentially behaviors, a texture is also a behavior, and it is capable of changing over time. This sample shows how a model's texture could be morphed from one image to another:

```
greenskin = import ("gskin.jpg");

brownskin = import ("bskin.jpg");

lizard3d = import ("lizard.wrl");

skin =opacity (1 / time, greenskin) over brownskin;

lizard = texture (skin, lizard3d);

lizardimage = renderedGeometry (lizard, defaultCamera);
```

There are two special kinds of geometries that are not visibly rendered: lights and spatially oriented sounds. Because these objects are of type geometry, they have the capability to be manipulated with full three-dimensional flexibility.

Lights come in four varieties, each with its own constructor function:

- ambientLight(type geometry) falls on all illuminated objects equally from a directionless source. Because ambient light is directionless, transformations have no visible result on this type.
- directionalLight(type geometry) is cast from an infinitely distant point along a given vector. By default, the direction vector lies along the -Z axis (into the screen), but you can rotate it into any desired orientation.
- pointLight(type geometry) radiates from a single point into all directions equally. The default location is at the origin (0, 0, 0), but it can be translated into any position.
- spotLight(type number * number * number -> geometry) is cast from a single point, filling a cone-shaped volume along a directional vector. The intensity with which a spotlight illuminates an

object drops off exponentially near the cone boundary, as specified by the three parameters to the constructor: fullcone, cutoff, and exponent. They describe the radius of full illumination, the radius where there is no illumination, and an exponent that governs the transition formula. By default, spotlights are directed along the -Z axis and are located at the origin.

The ActiveVRML language specification states that lights only illuminate the other geometries that they are aggregated with. This feature is not supported by many rendering engines, including Direct3D. That means that an instantiated light will influence all the objects in the current scene, whether or not they have been joined with the union function.

Light geometries can have customized color and attenuation characteristics. The function lightColor (type color * geometry ->geometry) is used to change the color of a single light, or all of the lights in a defined aggregate, from the default color of white. The way that light drops off over distance (if at all) is controlled with the function lightAttenuation, of type number * number * number * geometry -> geometry. The three number parameters, c, l, and q, are the coefficients of the equation 1 / (c + ld + qdd), that yields the intensity with which an object at distance d is illuminated. The default values are (1, 0, 0), which results in no attenuation at all.

Spatially oriented sound is a great feature that is implemented with a special geometry type created with the soundSource function (type sound -> geometry). The location of the sound can then be directly modified by transformations, or it can be aggregated with some other geometry object.

2D Types

Image objects are not the only types that you can manipulate with two-dimensional transformations. Points and vectors can also be defined, each of which has two components for describing X and Y coordinates. Even though these types have no visible appearance, they are useful tools for building complex behaviors.

The most common way to construct a 2D point is with the function point2Xy (type number * number -> point2), or a default point at the origin can be created with origin2 (type point2). Points may be added and subtracted, the separation between two points can be retrieved with distance (type point2 * point2 -> number), and a point can be transformed with the transformPoint2 function (type transform2 * point2 -> point2). The components of a point are extracted with xComponent (type point2 -> number) and yComponent (type point2 -> number).

Vectors also have an X and Y component, and they are constructed with the function vector2Xy (type number * number -> vector2). Many behaviors simply need vectors along the X or Y axis, so two special constructors are provided for this purpose, xVector2 and yVector2, each of type vector2. Vector functionality includes length (type vector2 -> number), normal (type vector2 -> vector2), and dot (vector2 * vector2 -> number), as well as the ability to add, subtract, and scale vector values. Transformations are applied with transformVector2, of type transform2 * vector2 -> vector2.

These 2D types have the common ability to be subjected to planar transformations, using functions that require the transform2 type object. Common transformations include translation, rotation, and shearing, or

any combination of these used together. For example, to scale an image using the values of a supplied vector, and then rotate it 90 degrees clockwise, this code would suffice:

```
scaledbird = tranformImage (scale (inputvector), parrotimage);

rotatedbird = transformImage (rotate (-1.570796), scaledbird);
```

The transformations could be combined into one declaration using the o function, which composes multiple transformations. The order in which the transformations is applied is important. This fragment has identical functionality to the previous one:

```
Transformedbird =

   transformImage (rotate (-1.570796) o

      scale (inputvector), parrotimage);
```

3D Types

There are 3D points, vectors, and transformations that correspond very closely with their 2D counterparts, both in name and in usage. For instance, the vector constructor vector3Xyz (type number * number * number -> vector3) simply has an additional parameter to describe the Z direction. Like 2D points and vectors, these are not visible but are used instead to manipulate geometry objects. The transform3 type is likewise very similar to a transform2 type, but extends translations, rotations, and scaling into 3 dimensions. The comprehensive sample given later in this chapter will focus heavily on these 3D types, so they will not be discussed any further here.

Camera Type

The camera type is used in conjunction with the renderedImage function to render a geometry object onto a 2D surface. This type of object is subject to three-dimensional transformations that determine the characteristics of the 3D-to-2D conversion.

You can think of the camera object as a projection plane through which an observer views a scene from a specific projection point. In a default camera, created with the defaultCamera function (type camera), the viewer's location is at (0, 0, 1) and the viewing orientation is along the -Z axis through the projection plane at z=0.

Scaling transformations applied to the X and Y axis of the projection plane can squeeze or lengthen a

resulting image, and scaling the distance between the projection point and the plane will give control of zoom. Transformations are applied with the transformCamera function, of type transform3 * camera -> camera.

Color Type

Both geometry and text types can have color properties, as defined by objects of the color type. You can create colors in standard varieties with constructor functions like red (type color) and magenta (type color). Customized colors can be made with colorRgb and colorHsl, both of which are type number * number * number -> color. The parameters refer to red-green-blue or hue-saturation-lightness, respectively. Individual components of an existing color can be extracted with functions like greenComponent (type color -> number) or saturationComponent (type color -> number).

Sound and Microphone Types

In ActiveVRML, you can associate sound objects with geometry to simulate spatial location, or you may simply define them in terms of the imported data. Importation and manipulation of sound objects has already been discussed in previous sections, but the function renderedSound (type geometry * microphone -> sound) must still be described.

The function renderedSound works in conjunction with the geometry function soundSource to define both the sound-producing and the sound-listening locations. The function result is a sound object that accurately represents this relationship.

The microphone object is created with the defaultMicrophone function (type microphone) at the origin (0, 0, 0), and can be transformed in three-dimensional space like a geometry object using transformMicrophone (type transform3 * microphone -> microphone).

Text Handling Types

One of the more powerful features that makes ActiveVRML useful for designing Web sites is the ability to draw and manipulate text using a concise but effective set of functionality. You can construct string objects from a supplied string or from a list of characters. They are then used to create text objects that control the actual rendering to an image, based on color and font-family information.

This example demonstrates the majority of the text handling functions:

```
counttext = numberToString (count, 0);
```

```
redtext = textColor (red, simpleText ("Count: ") & counttext);

sanstext = textFamily (sansSarifProportional, redtext);

textimage, textextent = renderedText (bold (sanstext));
```

The function numberToString (type number * number -> string) will convert the supplied number into a string, using the precision argument to control how many digits will appear after the decimal. The text object is created by the simpleText function (type string -> text) and has color applied with colorText (type color * text -> text). Although the specific font cannot be chosen yet with ActiveVRML, you can control the basic appearance by choosing one of three distinct font families—serifProportional, sansSerifProportional, or monospaced, as an argument for the textFamily function (type fontFamily * text -> text).

Program Structure

There is no sense of "program flow" when looking at an ActiveVRML script, as there is in most other languages. Instead, each object carries with it an individual sense of time and behavior. Although the value of an object may change over time, or in response to a predefined event, the object's definition never changes during its lifespan. Once all the relationships between defined objects have been set up, everything is let loose to run on its own. An ActiveVRML programmer never needs to know details like how often an expression is re-evaluated, or which objects need updating at some point in time—those complexities are left to the AVRML engine.

Declarations

Objects relate to each other in several flexible ways within the ActiveVRML language. Each object has an identifier and a value, which are declared once in the script. The value is an expression that can refer to both other objects and also the object being defined, recursively.

Functions can also be declared in ActiveVRML to define more complex relationships between objects than simple expressions could handle. They can have parameters for accepting values, and they can return one or more values as results. Here is one example:

```
halfdistance (point1, point2) = distance (point1, point2) / 2;
```

This example shows how multiple values may be returned:

```
vectorlen (vec) = length (vec), length (vec) / 2;

(fulllength, halflength) = vectorlen (vector2Xy (20, 30));
```

Expressions of the form if-then-else are allowed, which evaluate to one of two branches depending on the Boolean value of the if condition. Here is an example:

```
getbirdimage (val) =

   if val = 1 then

      parrotimage

   else

      ravenimage;
```

Depending on the value of the supplied parameter, the preceding snippet of code would return one of two image values. Both branches must always be present when using this expression form, to correctly handle the Boolean test.

Scope of Identifiers

The identifier name associated with an object has global scope by default, meaning that any expression within the script may refer to the object if it has been defined. To reuse a name, a mechanism must be introduced to provide local scoping. In ActiveVRML, that mechanism is an expression form using a let-in syntax.

This functionality allows local declarations to be made within the body of the let expression. These declarations are available to the expression found after the in keyword, but nowhere else in the script. This is an example of how the expression is used:

```
getposition (startx, starty) =

   let

      xpos = startx + time * 3;

      ypos = starty + time / 2;

   in
```

```
      point2Xy (xpos, ypos);

currentlocation = getposition (15, 20);
```

The two declarations that we wanted to make local are xpos and ypos. The point2 object named currentlocation is defined in terms of the function getposition, but the actual methods used by getposition are better off hidden from the rest of the script. Some alternate version of this function can now be declared that also uses xpos or ypos, and there would be no conflict.

In this case, some parameters are supplied to initialize the function's behavior. At the moment that currentlocation was first evaluated, the object's value would be (15, 20), the same as the supplied values. But after 2 seconds, the value of currentlocation would be (21, 21). Notice that the named parameters startx and starty are available to the internal declarations, and they would also be available to the in expression.

Time-Varying Behaviors

The concept of time has been covered in a variety of samples already, but there are some additional points that need further explanations. Consider this code fragment:

```
getimage =

   let

      if time > 5 then

         current = parrotimage;

      else

         current = ravenimage

   in

      current until predicate (time > 10) =>

         getimage;
```

The initial returned value of getimage will be the value of the expression current, which will evaluate to ravenimage at first, then after 5 seconds, it will become parrotimage. But after 10 seconds, the predicate event will cause a new expression to be evaluated. In this case the new expression is recursive, and by evaluating it, a new getimage object will be created with a local time of zero. The end result of this

declaration is a behavior whose value toggles between two values, keeping each for 5 seconds.

An interesting function called timeTransform can modify the way local time is calculated. It is of type a * number -> a, meaning that it can accept any type as a parameter, together with a number that must itself be time-varying. A copy of the named behavior is returned, with an internal clock tied to the supplied number. For instance, consider what would happen if the declaration in the preceding sample could be used like this:

```
quickimages = timeTransform (getimage, time * 4);
```

Now each resulting image value would only be displayed for 1.25 seconds at a time, because time itself would have a quadrupled rate. There are two rules for how time can be manipulated with this function—the resulting passage of time must always be positive, and it must always be increasing.

Reactive Behavior and Events

A key ingredient of ActiveVRML's flexibility is the ability to have behaviors that react to external events. The examples so far have demonstrated the until function together with a predicate generated event, as in:

```
bird = parrot until predicate (time > 5) => raven
```

The identifier bird is defined by the expression parrot until the predicate event is triggered, then it is defined by the expression on the right side of =>, which is raven.

Events can also be generated by user input with a keyboard or with the mouse. For example, the system event leftButtonPress is an event triggered by pressing the left mouse button. Here is how it could be implemented:

```
bird = parrot until

   leftButtonPress => raven |

   predicate (time > 5) => bird;
```

This sample also shows how to specify an alternate event, using the | operator. Whichever event occurs first is the one whose handler expression is evaluated. The value of bird will always be parrot until the user presses the left mouse button, and then the value will be raven for 5 seconds before switching back to parrot.

It is often desirable to know what the value of a behavior was at the time an event is triggered. This is

accomplished through use of the snapshot function, which will sample the value of an expression when the associated event fires. For example, to define a value that you only want updated every 10 seconds, you would write the following:

```
newsize = time * 1.1234;

size (oldsize) = oldsize until

   snapshot (newsize, predicate (time>10)) => size;
```

Although newsize is increasing continuously, the value of the size function only reflects the changes every time the predicate event is triggered. The updated value is used as an argument to the recursive redefinition of size, while time is reset to zero.

You can define events that are triggered by an external program, through the COM interface. Typically, this is an event that is tied to a script running in the parent HTML document. This kind of event can be declared with the importUnitEvent function, which requires a unique numeric identifier, for instance:

```
scriptevent = importUnitEvent (100);

bird = parrot until scriptevent => raven;
```

TIP

All external events are presented to ActiveVRML through a single interface, so the only way to identify individual events is with their associated numeric identifying code. For proper event handling to occur, these numbers must match in both the AVRML script and in the external script.

External events may also carry numeric or string data, and they would be declared with importNumberEvent or importStringEvent. Here is an example of how such data is acquired:

```
birdsize = importNumericEvent (100);

birdweight (size) = size * 10;

newweight (oldsize) = birdweight (oldsize) until

   birdsize => birdweight;
```

This may seem a bit confusing at first, because it is not clear how the imported data is being applied. The key point is that the data is implicitly used as an argument to birdweight after the event has been triggered. Type checking is performed here, as everywhere else, to make sure that the handler function

does indeed accept argument of the correct type to match the event.

A more compact way to represent the previous behavior is to use a special form of anonymous function declaration. This method provides a way of defining a function within the body of the event handler, without having to declare it separately. Here is a rewrite of the last sample:

```
birdsize = importNumericEvent (100);

newweight (oldsize) = oldsize * 10 until

   birdsize => function .x * 10;
```

User Input

ActiveVRML has a rich set of system events that are triggered by user-input through the mouse and keyboard. Several system objects are also available that continuously track the state of keys and buttons.

Mouse button events are: leftButtonDown, rightButtonDown, leftButtonUp, and rightButtonUp. The Boolean state of these buttons can also be tracked with leftButtonState and rightButtonState, which evaluate to TRUE if the button is currently being pressed. The position of the mouse is likewise tracked with mousePosition, which has a type of point2.

Similar events are tied to keypresses, and they react when specific keys change state. They are: keyDown, keyUp, and keyState. An argument is supplied to each of these events when they are declared, indicating the Virtual Key Code of the key to be watched. Some of the more common codes are:

- vkLeft
- vkRight
- vkUp
- vkDown
- vkReturn
- vkSpace
- vkEscape

An additional event, charEvent, does not take a Virtual Key Code as an argument, but instead it gives the Boolean state of a supplied ASCII character.

Advanced Sample

Now it's time to present a more complete sample that will really show off the power of the language. This script will demonstrate the implementation of a simple, yet engaging, game called CyberBall.

Microsoft includes a few eye-catching sample scripts with the ActiveVRML control, and they are well worth looking at to understand the true variety of designs that are possible. However, these scripts focus on manipulating two-dimensional images and sound, and they don't offer an in-depth look at 3D virtual environments. Another topic given little attention is AVRML's ability to interface with external scripts and HTML. CyberBall is written to emphasize these powerful capabilities and to show developers where some pitfalls still remain.

The concept behind CyberBall is hardly new. A player controls a "CyberBall" object as it slides around a rectangular arena. The aim is to use the CyberBall to knock a puck into a goal at one end of the arena, while avoiding the goal at the opposite end. The Arena, the CyberBall, and the puck are all imported VRML objects. An HTML interface provides game controls, score display, and control of dynamic camera positioning. Figure 17.2 shows how it all looks on screen.

Figure 17.2. CyberBall in action.

CyberBall Source Code

A look at this source code will reveal a lot of methods that should be familiar to you by now, but after the listing (see Listing 17.5) some of the more interesting parts will be examined in greater detail.

Listing 17.5. Cyberball.avr.

```
1: // ActiveVRML 1.0 ASCII

  2: // CyberBall Sample Script

  3:

  4: // First we create the arena, starting with the four corners

  5: corner = first (import ("cyl.wrl"));

  6: cornercolor = colorRgb (0.5, 0.5, 0.5);

  7: post = diffuseColor (cornercolor,
```

```
 8:    transformGeometry (scale (0.5, 0.5, 0.5), corner));
 9: corners = transformGeometry (translate (-15,0.20,-20), post) union
10:    transformGeometry (translate (-15, 0.20, 0), post) union
11:    transformGeometry (translate (-15, 0.20, 20), post) union
12:    transformGeometry (translate (15, 0.20, -20), post) union
13:    transformGeometry (translate (15, 0.20, 0), post) union
14:    transformGeometry (translate (15, 0.20, 20), post);
15:
16: // Now the goals
17: goal1 = diffuseColor (colorRgb (0.5, 0.5, 1.0),
18:    transformGeometry (scale (0.3, 0.5, 0.3), corner));
19: goal2 = diffuseColor (colorRgb (1.0, 0.5, 0.5),
20:    transformGeometry (scale (0.3, 0.5, 0.3), corner));
21: goals = transformGeometry (translate (-3, 0.20, -20), goal1) union
22:    transformGeometry (translate (3.0, 0.20, -20.0), goal1) union
23:    transformGeometry (translate (-3.0, 0.20, 20.0), goal2) union
24:    transformGeometry (translate (3.0, 0.20, 20.0), goal2);
25:
26: // Set up some side rails and end rails
27: rail = first (import ("stick.wrl"));
28: railcolor = colorRgb (0.5, 3.0, 0.5);
29: siderail = diffuseColor (railcolor, transformGeometry (
30:    rotate (yVector3, 1.57079) o scale (1.0, 0.3, 0.2), rail));
```

```
31: sides = transformGeometry (translate(-15, 0, -10), siderail) union
32:    transformGeometry (translate (15, 0, -10), siderail) union
33:    transformGeometry (translate (-15, 0, 10), siderail) union
34:    transformGeometry (translate (15, 0, 10), siderail);
35: endrail = diffuseColor (railcolor,
36:    transformGeometry (scale (0.6, 0.3, 0.2), rail));
37: ends = transformGeometry (translate (-9.0, 0, -20), endrail) union
38:    transformGeometry (translate (9.0, 0, -20.0), endrail) union
39:    transformGeometry (translate (-9.0, 0, 20.0), endrail) union
40:    transformGeometry (translate (9.0, 0, 20.0), endrail);
41: rails = sides union ends;
42:
43: // Create some lights
44: lights = lightColor (colorRgb (0.5, 0.5, 0.5),
45:    transformGeometry (rotate (xVector3, -0.1), directionalLight)
46:    union
47:    transformGeometry (rotate (xVector3, 3.24), directionalLight));
48:
49: // Put it all together and define a background
50: arena = corners union goals union rails union lights;
51: backgnd = first (import ("backgnd.gif"));
52:
53: // Build a puck
```

```
54: puck, pmin, pmax = import ("puck.wrl");
55: puckradius = xComponent (smax) * 0.05;
56: correctedpuck = transformGeometry (scale(0.05, 0.05, 0.05), puck);
57: coloredpuck = diffuseColor (red, correctedpuck);
58:
59: // Build a player
60: socket, smin, smax = import ("socket.wrl");
61: playerradius = xComponent (smax) * 0.1;
62: correctedsocket = transformGeometry (rotate (xVector3, 1.57079) o
63:    scale(0.1, 0.1, 0.1), socket);
64: coloredsocket = diffuseColor (blue, correctedsocket);
65: ball = first (import ("ball.wrl"));
66: correctedball = transformGeometry (translate (0, 0.6, 0) o
67:    scale (1.3, 1.3, 1.3), ball);
68: spinningball = transformGeometry (rotate (yVector3, time * 3),
69:    correctedball);
70: coloredball = diffuseColor (red, spinningball);
71: player = coloredsocket union coloredball;
72:
73: // Load some sounds
74: winner = first (import ("winner.wav"));
75: loser = first (import ("loser.wav"));
76: wallhit = first (import ("wallhit.wav"));
```

```
77: puckhit = first (import ("puckhit.wav"));
78:
79: // Helper function to restrict value to a specified range
80: clamp (val, min, max) = if (val < min) then min
81:     else if (val > max) then max
82:     else val;
83:
84: // User input events
85: keyleft = keyState (vkLeft);
86: keyright = keyState (vkRight);
87: keyforward = keyState (vkUp);
88: keyback = keyState (vkDown);
89:
90: // Handle player motion
91: playermover (pos0 : vector3, vel0 : vector3) =
92:    let
93:       rebound (pos : vector3, vel : vector3, snd) =
94:          let
95:             // Dampen function slows player down over time
96:             dampen (oldvel : vector3) =
97:                vector3Xyz (xComponent (oldvel) * 0.5 / time,
98:                   0, zComponent (oldvel) * 0.5 / time);
99:
```

```
100:              // Calculate force applied by user control
101:              accelstrength = 5;
102:              forwardaccel =
103:                 if keyforward then
104:                    vector3Xyz (0, 0, -accelstrength)
105:                 else if keyback
106:                    then vector3Xyz (0, 0, accelstrength)
107:                 else zeroVector3;
108:              sideaccel =
109:                 if keyleft
110:                    then vector3Xyz (-accelstrength, 0, 0)
111:                 else if keyright
112:                    then vector3Xyz (accelstrength, 0, 0)
113:                 else zeroVector3;
114:              pushvel = integral (forwardaccel + sideaccel);
115:
116:              // Calculate new unclamped velocity
117:              newvel = dampen (vel) + pushvel;
118:              magnitude = length (newvel);
119:
120:              // Clamp velocity
121:              newmagnitude = clamp (magnitude, -4, 4);
122:              velocity = normal (newvel) * newmagnitude;
```

```
123:
124:            // Calculate new position
125:            newpos = pos + integral (velocity);
126:            xpos = (xComponent (newpos));
127:            zpos = (zComponent (newpos));
128:
129:            // Handle side wall collisions
130:            sideextent = 15 - playerradius;
131:            sidecollide = predicate ((xpos < -sideextent) or
132:               (xpos > sideextent));
133:            sidenewvel = vector3Xyz (xComponent (velocity) * -1,
134:               yComponent (velocity), zComponent (velocity));
135:            sidenewpos = vector3Xyz (clamp (xComponent (newpos),
136:               -sideextent, sideextent), 0, zComponent (newpos));
137:
138:            // Handle end wall collisions
139:            endextent = 20 - playerradius;
140:            endcollide = predicate ((zpos < -endextent) or
141:               (zpos > endextent));
142:            endnewvel = vector3Xyz (xComponent(velocity),
143:               yComponent(velocity), zComponent(velocity) * -1);
144:            endnewpos = vector3Xyz (xComponent (newpos), 0,
145:               clamp (zComponent(newpos), -endextent, endextent));
```

```
146:
147:          in
148:             (newpos, velocity, snd) until
149:                snapshot ((sidenewpos, sidenewvel, wallhit),
150:                   sidecollide) => rebound |
151:                snapshot ((endnewpos, endnewvel, wallhit),
152:                   endcollide) => rebound;
153:    in
154:       rebound (pos0, vel0, silence);
155:
156:
157: // Handle puck motion
158: puckmover (pos0 : vector3, vel0, ppos0, pvel0, score0) =
159:    let
160:       rebound (pos : vector3, vel, ppos1, pvel1, snd, oldscore) =
161:          let
162:             // Dampen function slows puck down over time
163:             dampen (oldvel : vector3) =
164:                vector3Xyz (xComponent (oldvel) * 0.5 / time,
165:                   0, zComponent (oldvel) * 0.5 / time);
166:             velocity = dampen (vel);
167:
168:             // Calculate new position
```

```
169:              newpos = pos + integral (velocity);
170:              xpos = (xComponent (newpos));
171:              zpos = (zComponent (newpos));
172:
173:              // Handle side wall collisions
174:              sideextent = 15 - puckradius;
175:              sidecollide = predicate ((xpos < -sideextent) or
176:                 (xpos > sideextent));
177:              sidenewvel = vector3Xyz (xComponent (velocity) * -1,
178:                 0, zComponent (velocity));
179:              sidenewpos = vector3Xyz (clamp (xComponent (newpos),
180:                 -sideextent, sideextent), 0, zComponent (newpos));
181:
182:              // Handle end wall collisions
183:              endextent = 20 - puckradius;
184:              endcollide = predicate (((zpos < -endextent) or
185:                 (zpos > endextent)) and ((xpos<-3) or (xpos>3)));
186:              endnewvel = vector3Xyz (xComponent(velocity),
187:                 0, zComponent(velocity) * -1);
188:              endnewpos = vector3Xyz (xComponent (newpos), 0,
189:                 clamp (zComponent(newpos), -endextent, endextent));
190:
191:              // Goal-keeping functions
```

```
192:              EVENTSCORE = 200;
193:              goalnewvel = zeroVector3;
194:              goalnewpos = zeroVector3;
195:              goodscore = oldscore + 1;
196:              badscore = oldscore - 1;
197:
198:              // Handle good goal collisions
199:              goal1event = predicate ((zpos < -endextent) and
200:                 (xpos > -3) and (xpos < 3));
201:              collide1data = snapshot (goodscore, goal1event);
202:              goal1collide = exportEvent (collide1data, EVENTSCORE);
203:
204:              // Handle bad goal collisions
205:              goal2event = predicate ((zpos > endextent) and
206:                 (xpos > -3) and (xpos < 3));
207:              collide2data = snapshot (badscore, goal2event);
208:              goal2collide = exportEvent (collide2data, EVENTSCORE);
209:
210:              // Determine player's position, velocity, and sound
211:              (ppos, pvel, psound) = playermover (ppos1, pvel1);
212:              mixsnd = snd mix psound;
213:
214:              // Handle player collisions
```

```
215:             relativevector = newpos - ppos;
216:             playercollide = predicate (length (relativevector) <
217:                (playerradius + puckradius));
218:             oldenergy = length (velocity);
219:
220:             // Clamp velocity
221:             newvel = relativevector * oldenergy + velocity + pvel;
222:             magnitude = length (newvel);
223:             newmagnitude = clamp (magnitude, -4, 4);
224:             phitnewvel = normal (newvel) * newmagnitude;
225:             phitnewpos = newpos + normal (phitnewvel) *
226:                (playerradius + puckradius);
227:
228:       in
229:          (newpos, velocity, ppos, pvel, mixsnd, oldscore) until
230:             snapshot ((sidenewpos, sidenewvel,ppos, pvel, wallhit,
231:                oldscore), sidecollide) => rebound |
232:             snapshot ((endnewpos, endnewvel,ppos, pvel, wallhit,
233:                oldscore), endcollide) => rebound |
234:             snapshot ((goalnewpos,goalnewvel,ppos, pvel, winner,
235:                goodscore), goal1collide) => rebound |
236:             snapshot ((goalnewpos,goalnewvel,ppos, pvel, loser,
237:                badscore), goal2collide) => rebound |
```

```
238:             snapshot ((phitnewpos,phitnewvel,ppos, pvel, puckhit,
239:                oldscore), playercollide) => rebound;
240:    in
241:       rebound (pos0, vel0, ppos0, pvel0, silence, score0);
242:
243: (puckpos, puckvelocity, playerpos, playervelocity, sounds, score)=
244:    puckmover (vector3Xyz (0, 0, -5), vector3Xyz (-2, 0, 0),
245:       vector3Xyz (-10, 0, -6), vector3Xyz (-2, 0, 0), 0);
246:
247: // Apply all positional changes to the player and the puck
248: puckmotion = translate (puckpos) o rotate (yVector3, time * 2.9);
249: activepuck = transformGeometry (puckmotion, coloredpuck);
250: playermotion = translate (playerpos) o rotate (yVector3, time);
251: activeplayer = transformGeometry (playermotion, player);
252:
253: // HTML User Input Events
254: EVENTSTART = 100;
255: EVENTSTATIC = 101;
256: EVENTORBITAL = 102;
257: EVENTTRACKING = 103;
258: extstartevent = importUnitEvent (EVENTSTART);
259: extstaticevent = importUnitEvent (EVENTSTATIC);
260: extorbitalevent = importUnitEvent (EVENTORBITAL);
```

```
261: exttrackingevent = importUnitEvent (EVENTTRACKING);
262:
263: // Define the linear tracking and the orbital cameras
264: staticxform = rotate (xVector3, -0.25) o scale (1, 1, 0.3) o
265:    translate (0, 0.5, 200 + zComponent (playerpos));
266: orbitxform = rotate (yVector3, time / 4) o
267:    rotate (xVector3, -0.25) o scale (1, 1, 0.3) o
268:    translate (0, 0.5, 200);
269:
270: // Determine current camera transformation
271: cameraxform (xform) = xform until
272:    extstaticevent => cameraxform (staticxform) |
273:    extorbitalevent => cameraxform (orbitxform);
274: currentxform = cameraxform (staticxform);
275: camera = transformCamera (currentxform, defaultCamera);
276:
277: // Use camera transformation to position headlight
278: headlight = transformGeometry (currentxform,
279:    lightColor (colorRgb (0.5, 0.5, 0.5), directionalLight));
280:
281: // Compute cropped area
282: topright = point2Xy (xComponent (viewerUpperRight),
283:    yComponent (viewerUpperRight));
```

```
284: bottomleft = point2Xy (-xComponent (viewerUpperRight),

285:    -yComponent (viewerUpperRight));

286:

287: // Output everything

288: totalsounds = sounds;

289: totalobjects = arena union activepuck union activeplayer;

290: totalimage = crop (bottomleft, topright,

291:    renderedImage (totalobjects union headlight, camera));

292: model = (totalimage over backgnd, totalsounds, totalsounds)

293:    until extstartevent => model;
```

Description of the Sample

The first thing done in the script is to create the various elements of the arena, the cyberball, and the puck. All of these pieces use imported VRML objects to form their basic geometries. Then the behaviors of moving objects are defined, together with the sounds they produce. The final required parts define the camera behaviors, light placement, and user input.

Lines 1-50 focus on building the arena. The required components are: six cylindrical corner posts, four cylindrical goal posts, and eight rectangular side rails. The basic process is to import the raw geometry data, apply a material with diffuseColor, use transformGeometry to spatially locate individual elements, and join it all together with the union function. Stationary lights are also created and placed. The final result, in Line 50, is a single object named arena that is the composition of all the previous diverse geometries.

The next step is to build the puck and a cyberball player. When the geometries for these objects are imported, you will notice (in Lines 54 and 60) that extents are also acquired, which define the enclosing volumes of each shape. Later on, collisions will be detected and this information will be necessary to determine when shape boundaries intersect. Transformations are applied to the puck and player to modify their sizes and orientations, but not yet to indicate location. The puck will end up looking like a red aspirin, and the cyberball itself is composed of a large continuously spinning sphere enclosed by a blue stationary torus.

Four keyboard state objects are defined in Lines 84-88. These objects are tied to the arrow keys and are used to control the cyberball's movement behavior.

The behavior of the player is handled by the function playermover, which accepts an initial position and velocity and returns the current position, velocity, and sound. This function is somewhat long (Lines 91-154) and complicated, so it will be described in detail.

At first it may seem confusing to see the multiple use of the let operator, which is used to limit the scope of names. The idea is that there is another, embedded, function called rebound that actually does the work. What does this buy you? When the player collides with an arena wall, several things will happen—a sound will be played, the velocity vector will be substantially changed, and the position will be constrained to keep the player within the "physical" bounds of the arena area. These actions are all similar in the sense that they are one-shot behaviors; at all other times, the player's behavior is strictly time-varying. What happens after such a collision is that the one-shot behaviors are evaluated and then the results are used to rebuild the entire rebound function with a recursive redefinition, resetting time along the way.

It is necessary to reset time, because the player's position is calculated as the integral of the current velocity, using the integral function (Line 125). The discontinuous change in velocity after a collision would result in an invalid position, so calculations are started fresh each time.

The keys being watched for user input are used to apply acceleration forces to the player's cyberball. This acceleration is constant and results in velocity changes determined by another use of the integral function (Lines 101-114). These forces are easy to implement, because they are only applied in fixed directions along the X and Z axis. Velocity changes are also imposed by a damping function that simulates friction, but this is slightly more complicated to program (see Lines 96-98) because the negative acceleration is always directed along the cyberball's current velocity vector. The final factor in computing the current velocity is to limit the player's speed with the clamp function, which is declared on Line 80 and used on Line 121.

The velocity is used to determine where the next position will be, assuming that nothing gets in the way (like a wall). Two events are defined that will trigger when the player crosses the side or end boundaries. New position and velocity objects are defined that assume a new collision has just taken place, but it is important to realize that these definitions are only referred to when the actual event occurs.

These collision events, sidecollide and endcollide, are watched as part of the behavior described in Lines 148-152. When, for instance, sidecollide is triggered, a snapshot is taken of the behaviors sidenewpos and sidenewvel. These values are then used, together with the sound wallhit, as arguments for the new rebound function.

The behavior of the puck is declared in puckmover (Lines 158-241). This behavior uses the same general approach as playermover, but it has to react with several additional events. Not only must the puck rebound from wall collisions, but it must also detect and respond to collisions with the cyberball, and watch for goals to be scored. What it does not have to do is respond to user input, so the only way to get it moving is to bump it.

When a goal is scored, the puck is placed back at the center of the arena. Depending on which goal the puck was pushed through, the score is incremented or decremented and used as data for triggering an external event using the exportEvent function (Lines 202 and 208). This data becomes associated with a specific event by use of the snapshot function, and it is used by an external script to keep track of the current score.

In Line 211, playermover is actually invoked, and the player's state is determined. Any sounds that the player may be generating are now mixed with the puck's own sounds at Line 212, using the mix function. The location returned by playermover is used to check for any possible collision between the cyberball and the puck.

If a player collision is detected, the velocity of the puck will be modified using the value of the relative vector between the two objects as well as the velocity vector of the cyberball. This produces a more or less realistic rebounding effect.

The positions, velocities, and sounds of both the cyberball and the puck are brought together in Lines 243-245, with the call to puckmover. These values are now applied to the actual geometry objects to position them within the 3D environment.

Two other objects must also be positioned—the camera and a headlight. The camera has two separate behaviors, under the control of an external event (Lines 264-275). By default, it is positioned at one end of the arena and only moves in a linear fashion, back and forth along the Z axis, to maintain a fixed distance from the cyberball. But the camera can also be placed into an orbital mode, where it continuously circles the arena's perimeter while staying focused at the center. A headlight is created (Lines 278-279), which shares the same transformation as the camera, so that there is sufficient light no matter which direction is being observed.

In Lines 282-285, the two-dimensional extents of the display screen are calculated, which will be used to crop the final image. This cropping is not strictly necessary, but the result of rendering geometry objects is an image of infinite extent. By restricting the image to a given rectangular area, some performance gains may be realized.

The final twist is to the model declaration, which provides the expression used to output the entire scene. By reacting to the external event extstartevent, the entire scene can essentially be restarted from the beginning with a simple recursive definition.

Accompanying HTML Code

The CyberBall sample relies on bidirectional events to manage some areas of its functionality. Those events are specified in the parent HTML file—within a block of Visual Basic Script that is tied to some visible controls, as seen in Listing 17.6.

There are three named buttons that generate external events for the ActiveVRML script, and a text display whose output value is controlled by an internally generated event that carries associated data. The rest of the document can be recognized as standard HTML.

Listing 17.6. CYBERBALL.HTML.

```
<HTML>

<HEAD><TITLE>CyberBall Sample Page</TITLE></HEAD>
```

```
<BODY BGCOLOR=WHITE><CENTER>
<FONT SIZE=4><B>CyberBall ActiveVRML Sample Page</B></FONT>
<BR><BR>
<OBJECT
   ID="AVRCtrl"
   CLASSID="clsid:{389C2960-3640-11CF-9294-00AA00B8A733}"
   WIDTH=512 HEIGHT=256>
   <PARAM NAME="DataPath" VALUE="cyberball.avr">
   <PARAM NAME="Expression" VALUE="model">
   <PARAM NAME="Border" VALUE=TRUE>
</OBJECT><BR><BR>
<TABLE BORDER=1 WIDTH=400 CELLPADDING=10>
<TR><TH>Game Controls</TH><TH>Camera Controls</TH></TR>
<TR><TD VALIGN=CENTER ALIGN=CENTER>
<FONT FACE="ARIAL,HELVETICA" SIZE=3><B>Score</B></FONT>
<INPUT NAME=Score VALUE="0" SIZE=4,1><BR>
<INPUT NAME=Start TYPE=BUTTON VALUE="Start New Game"></TD>
<TD VALIGN=CENTER ALIGN=CENTER>
<INPUT NAME=Static TYPE=BUTTON VALUE="Static"><BR>
<INPUT NAME=Orbital TYPE=BUTTON VALUE="Orbital">
</TD></TR></TABLE><BR>
<TABLE WIDTH=500>
<TR><TH VALIGN=TOP>1:</TH><TD VALIGN=TOP>
```

```
<FONT SIZE=3>
Use the arrow keys to control your CyberBall. If you don't get
a response, try clicking first in the display window.
</FONT><BR>
</TD></TR>
<TR><TH VALIGN=TOP>2:</TH><TD VALIGN=TOP>
<FONT SIZE=3>
Knock the puck into the Blue goal to score a point,
avoid the Red goal or lose a point.
</FONT><BR>
</TD></TR>
</TABLE>
<SCRIPT LANGUAGE="VBScript"><!--
   sub Start_onClick
      EVENTSTART = 100
      AVRCtrl.FireImportedEvent (EVENTSTART)
      Score.value = "0"
   End sub
   sub Static_onClick
      EVENTSTATIC = 101
      AVRCtrl.FireImportedEvent (EVENTSTATIC)
   End sub
   sub Orbital_onClick
```

```
      EVENTORBITAL = 102
      AVRCtrl.FireImportedEvent (EVENTORBITAL)
   End sub
   sub AVRCtrl_ActiveVRMLEvent (EventID, Param)
      EVENTSCORE = 200
      If EventID = EVENTSCORE Then Score.value = Param
   End sub
--></SCRIPT>
</CENTER></BODY>
</HTML>
```

Summary

The descriptions and samples that have been presented in this chapter only scratch the surface of the creative possibilities opened up by the ActiveVRML language. The current implementation of Microsoft's AVRML control is amazingly well done for an initial test release, but it will continue to mature along with the language itself. Even in its early form, this control ranks among the most innovative and flexible technologies available for animating a Web site.

Chapter 18

OLE Document Objects

by Laurent Poulain

The specifications of *OLE Document Objects* (*DocObjects*, for short) evolved from existing OLE 2.0 specifications. However, DocObjects are more than just a new technology based on OLE 2.0 (like ActiveX controls or OCX); they represent the evolution of OLE's philosophy, and are a new breakthrough for document embedding. Microsoft intends to enforce these specifications in every document application on Windows—from Word to Internet Explorer (both of which, by the way, already support DocObjects).

Microsoft wants its upcoming Internet-oriented Windows 97 (also known as *Nashville*) desktop to be document oriented. That is, Microsoft wants every document from every application to be able to be inserted into a *container* (Office Binder, Internet Explorer,[el]).

What's New with DocObjects?

Because DocObjects are an evolution of OLE's document embedding, this chapter examines the major releases of this technology and what each release brings to document-embedding technology.

OLE 1.0

Although the first implementation of OLE (OLE 1.0) provides document-embedding capabilities, it isn't very satisfying from a visual standpoint. For example, every time a user wants to edit (create or modify) a document embedded in an OLE 1.0

container application, the user must edit the document in an external corresponding application (called the *server application*) that has been loaded by the container application (see Figures 18.1 and 18.2). When the user finishes, he or she closes the OLE server application to return to his or her original application. Although OLE 1.0 works perfectly from a technical standpoint, having to pass from one application frame window to another doesn't really create the visual impression of embedded document.

Figure 18.1. Viewing an OLE 1.0 embedding document in Microsoft Word.

Figure 18.2. Editing an OLE 1.0 embedded document. The OLE server application window (in this case Microsoft Paint) pops up and lets the user edit the embedded document.

OLE 2.0

OLE 2.0 addresses the visual issues of OLE 1.0. When users edit an embedded document using OLE 2.0, they can edit the embedded document within the boundaries of their current frame window. The server application is still loaded into memory, but users need not jump from the container application to the server application; indeed, some pieces of the container document are visible outside the boundaries of the embedded document. In addition, changes to OLE 2.0's environment (such as the menu bar, tool bar, and so on) have been implemented (see Figures 18.3 and 18.4).

Figure 18.3. Viewing an OLE 2.0 embedded document. So far, no change from OLE 1.0.

Figure 18.4. Editing an OLE 2.0 embedded document. The user edits the embedded document directly within the container frame window. The menu bar and toolbar have changed, but the frame window and the container document outside the embedded document's boundaries remain unchanged.

Although OLE 2.0 implements several useful changes, there remains room for improvement. OLE 2.0 lacks the capability to hide the embedded side of an embedded document. Indeed, an external, embedded document is bound to a rectangle within a native container document. A document window cannot only contain the external document. It must first contain a native document that will contain the external one. As a result, one can immediately recognize an external document. For example, Winword 6 can embed an Excel 5 graph within a Word document, but Winword cannot have a document window containing only the Excel chart (in other word, the Excel chart cannot be full-scaled).. Furthermore, the embedded application cannot control the frame window. For example, the Excel chart cannot change the type of view of the Word document (that is, normal mode, page mode) and don't have access to Word's save and print commands..

DocObjects

OLE DocObjects were designed to make the user forget that an embedded document is embedded. A DocObject can be embedded in a container application even if it is not within a native container document; it can control the frame window and can be viewed full-scale. In other words, a DocObject behaves like a native document.

Some DocObject applications already exist. Office 95 Binder and Internet Explorer 3.0 are examples of DocObject containers, and Office 95 applications, such as Word 95, Excel 95, and PowerPoint 95, are examples of DocObject server applications.

DocObject Examples

What are the benefits of DocObjects? What good is a DocObject container? After all, OLE 2.0 isn't so bad. If you want to edit a full-scale document, why not do it by loading the corresponding application? To answer these questions, let's take a look at Office Binder and Internet Explorer 3.0. Both are DocObject containers, so both allow full-scale use of DocObject applications

within their frame windows.

Office Binder, shipped with Microsoft Office 95, enables you to create catalogs of Microsoft Office documents. Each catalog is a collection of DocObjects (see Figure 18.5). The left pane of Office Binder represents all documents contained in the catalog. When you click a document, the right pane displays that document using the appropriate application interface (if it is a Word document, the right pane displays the Word pull-down menu, icon bars, and so on); you can then edit this document. In Microsoft Office Binder you can edit a set of documents from different applications. When you change from a Word document to an Excel document, the right pane automatically displays the Excel interface. Except for being able to see Microsoft Office Binder's left pane, you really have the impression of using the actual desktop application.

In Figure 18.5 the left pane represents the documents of the catalog (here, one Excel document and one Internet Explorer document). The right pane displays the selected document with its corresponding application interface.

Figure 18.5. The Office Binder window.

Internet Explorer 3.0 enables you to embed and display DocObjects just like it displays HTML pages (see Figure 18.6). That might seem odd for a Web browser, but when you consider the Internet-oriented, intranet-oriented, and document-oriented desktop policy found in Windows 97, it makes more sense. In Windows 97, Internet Explorer (the Web browser) and Microsoft Explorer (the file manager) are merged into one unique Explorer application. Therefore, local documents are accessed the same way as Internet HTML pages, and every document displayed within its frame window is seen as a DocObject by Internet Explorer (either the current Internet Explorer and the future Windows 97 Explorer). When Internet Explorer loads a document, it asks the operating system (Windows 95 or Windows NT) for the corresponding server application (that is, the application that manages the required document). If the server application isn't a DocObject server, Internet Explorer calls it and asks it to display the document. Otherwise, IE acts as a DocObject container, interacts with the DocObject server, and displays the document within its frame window. When dealing with HTML pages, IE just uses its embedded HTML DocObject server (unless .HTM/.HTML files are bound to another application, say Netscape Navigator, in which case IE will load Netscape Navigator. However, it much reduces IE's interest).It makes sense, and then, for a Web Browser to access documents other than HTML pages from the local hard disk, from a corporate intranet, or from the Internet (if the corresponding DocObject server application isn't remote), and to display them as native documents. That's why ActiveX technology includes hyperlinking features based on DocObject technology.

Figure 18.6. Internet Explorer 3.0 displaying a Word document instead of a regular HTML page.

A generic DocObject container (a combination of Office Binder and Internet Explorer) could feasibly be provided with the operating system, and would be the primary (if not the only) user interface.

DocObject Mechanics

To understand how DocObjects work, you must understand their mechanics. The DocObject model adds a few things to the OLE model from which it has evolved. Because DocObjects have evolved from OLE 2.0, programmers can turn existing OLE-compliant applications into DocObject-compliant applications without having to rewrite the applications.

DocObject Components

DocObjects use the same client/server architecture as OLE. A DocObject server application provides the document object, and a DocObject client, or *container*, embeds the DocObject and displays it in its frame window.

The DocObject server application provides the document object on the DocObject container's request. The DocObject server application also manages the document, implementing document-specific tasks such as storing and printing. Following the Microsoft Foundation Classes document/view architecture, the DocObject server application is composed of two parts:

- Document—This part manages all data that represents the document, and manages related operations, such as storage.

The document part is responsible for creating views.

- View—This part manages the views as well as related operations, such as printing. There can be several views for one document. Every view has an associated DocObject container view site.

The DocObject client, or container, embeds the DocObject within its frame window. The client, or container, is composed of several elements:

- Document site—This is the same as the client site object found in OLE documents. It contains every regular OLE 2.0 interfaces, plus an extra DocObject-specific interface. The document site contains all the view sites.
- View site—This is associated with a server application's view. The server application's view manages the display of the DocObject (that is, the screen space where the whole application will be displayed), whereas the view site manages the display space (that is, the screen space where the document will be displayed). A view site can be seen as a view frame (HWND type) or a view port (a rectangular area in the view frame).
- Frame—This manages the frame window of the container.

DocObject Interfaces

The document site, view site and frame only communicate through interfaces. An *interface* is a set of functions. From an implementation point of view, an interface is an instance of a C++ class whose member functions are the interface functions. For further information about interface implementation, see the section later in this chapter titled "Interface Implementation."

Some interfaces found in DocObjects already exist in OLE 2.0, so you won't have to write them in programs that support OLE 2.0 (OLE 2.0's IOleObject interface, however, will have to be slightly modified). DocObjects bring a set of new interfaces; some must be implemented, but some are optional.

How Everything Works Together

DocObject mechanics might seem fuzzy because of the number of function calls from the container to the server application and vice-versa. Because of the potential for confusion, I will detail some of DocObject's common procedures.

Initialization

Initialization occurs when the user decides to embed a DocObject. The process begins as standard OLE 2.0 requests, but ends as DocObject-specific processing.

- When the user (of the container) gives the order to embed a DocObject, the container calls the corresponding DocObject server application's IOleObject::DoVerb() interface member function. This interface is OLE-2.0 based. So far, this is normal OLE 2.0 embedding.
- The DocObject server application's IOleObject::DoVerb() is actually modified to perform a different process if it recognizes a DocObject client. If the client is DocObject-compliant and IOleObject::DoVerb() is passed either a OLEIVERB_SHOW or OLEIVERB_OPEN message, the server application calls the container's IOleInplaceSite::ActivateMe() interface member function.
- The IOleInplaceSite::ActivateMe() function activates the container. This function can ask the server application (the document part of the container) its IOleDocument interface through its IUnknown::QueryInterface() member function. The function can then create a view by using the server's IOleDocument interface.
- Now that the container has a memory pointer to the server application's IOleDocument interface, it can talk with the server application when it needs to.

Creating Views

The container manages the server application's view. To do so, the container requires the server's views' IOleDocumentView interface to create a new view or to manage an existing one. The following describes what happens when the container asks the server application to create a new view:

1. 1. The container calls the server application's IOleDocument::CreateView() interface member function.
2. 2. This function creates the view and returns the view's IOleDocumentView interface to the container.
3. 3. The container can then associate the created view with a container's view site (in order to work, a server application's view must be associated with a container's view site). To do so, the server application's view's IOleDocumentView::SetInPlaceSite() function is called, which passes the IOleInPlaceSite interface from one of the server application's view sites to the container.
4. 4. The server application's view knows its associated container's view site through its IOleInPlaceSite interface.

Enumerating Views

To retrieve an existing view's IOleDocumentView interface, the container must enumerate the existing views. The process for this follows:

1. 1. The container calls the server application's IOleDocument::EnumViews() interface member function.
2. 2. This server application function creates an *enumerator* to enumerate the existing views. This enumerator can be controlled via its IEnumDocumentViews interface, which is returned to the container.
3. 3. The container can then browse the existing views and select one (or more) that interests it through the server application's enumerator IEnumDocumentViews interface.
4. 4. The server application's enumerator then sends the selected views' IOleDocumentView interfaces to the container.
5. 5. The container can manage the selected views through their IOleDocumentView interfaces.

Interface Implementation

As you have seen, DocObject programming requires the implementation of several interfaces. You will now take a look at how interfaces are created.

An interface is implemented in several steps, the first of which is to declare the interface in the definition of its class:

```
class CMyClass: public CBaseClass

{

public:

    BEGIN_INTERFACE_PART(MyDocument, IMyInterface)

        INIT_INTERFACE_PART(CMyClass, MyDocument)
```

```
        STDMETHOD(Create)(int);

    END_INTERFACE_PART(MyDocument)

    ... // all the other interfaces

    DECLARE_INTERFACE_MAP()

};
```

In this example, the CMyClass class contains the IMyInterface interface. Notice the presence of the MyDocument name. This name represents the class that contains the interface implementation. Indeed, an implicit XMyDocument class is created and contains all the interface functions through its own member functions. Also, the member variable m_xMyDocument of type IMyInterface is implicitly declared in the CMyClass class. This variable allows a CMyClass instance to access its IMyInterface interface. Both XMyDocument class and m_xMyDocument variables are automatically created by the MFC macros. Their declaration is therefore not visible in the code.

The second step is to implement the declared interface:

```
IMPLEMENT_DYNAMIC(CMyClass, CBaseClass)

BEGIN_MESSAGE_MAP(CMyClass, CBaseClass)

    //{{AFX_MSG_MAP(CMyClass)

        // NOTE—the ClassWizard will add

        // and remove mapping macros here.

    //}}AFX_MSG_MAP

END_MESSAGE_MAP()

BEGIN_INTERFACE_MAP(CMyClass, CBaseClass)

   INTERFACE_PART(CMyClass, IID_IMyInterface, MyDocument)

END_INTERFACE_MAP()

STDMETHODIMP CMyClass::XMyDocument::Create(int n)

{

    METHOD_PROLOGUE_EX(CMyClass, MyDocument)

    ...

}
```

All interface functions always return a HRESULT variable. This variable describes the state of the operation (whether or not it succeeded, and if not, why). Because no output is directly returned, interface functions have two types of arguments: input and output. Input arguments are passed normally. Output arguments are actually pointers to a memory location where the function can write the output.

All interfaces are derived from the IUnknown base class. This class represents an unknown interface and implements the

virtual IUnknown::QueryInterface() member function, which is the way to retrieve an object's interface. This function is used to determine whether an object supports a given interface. For example, a server application's document object wants to get the container's document site IOleDocumentSite interface. The server application's document object must:

1. 1. Get the container's document site's IOleClientSite interface by calling the application server's DocObjects IOleDocument::GetClientSite() interface member function. Say the DocObjectt stores the interface pointer in m_xDocumentSite.
2. 2. The DocObject then calls m_xDocumentSite's QueryInterface() member function and asks the document site whether it supports the IOleDocumentSite interface(if yes, its pointer is stored in m_xOleDocumentSite):

```
m_xDocumentSite->QueryInterface(IID_IOleDocumentSite,

                                (void **)&m_xOleDocumentSite);
```

This process is common with DocObjects. Because DocObject components are only seen through their interfaces, one component will know the existence of another component only if it has a pointer to (one of) its interface. In the rest of the chapter, the term *passing an interface* means passing the interface's pointer. This term is used for easier reading.

DocObject Server Application Implementation

Implementing a DocObject server application requires writing several non-OLE features. The biggest part of a DocObject is the set of interfaces it supports, including some OLE 2.0 interfaces that don't need to be re-written if the server application already uses OLE 2.0. However, some interfaces are new DocObject interfaces. Following is a list of the DocObject-specific interfaces:

- IOleDocument interface (document part)—contains information about DocObjects. A DocObject allows the creation of views through IOleDocument.
- IOleDocumentView interface (views part)—view management. Allows modification or retrieval of some of the view's properties (such as the associated view site), to clone a view, and other miscellaneous operations.
- IEnumOleDocumentViews interface (optional)—views enumeration. Allows enumeration of every existing views.
- IOleCommandTarget interface (optional)—command target. Allows the DocObject to use some of the DocObject container's functions such as Save, Print, and so on. See the section later in this chapter titled "Command Target."
- IPrint interface (optional)—programmatic printing. See the section later in this chapter titled "Programmatic Printing."

In order to implement a server application, you must:

- Include the DocObject server application interfaces found in your COleServerDoc derived class. The IOleDocument and IOleDocumentView interfaces must be implemented. The OLE 2.0 IOleObject interface must be slightly modified. The complete description of these interfaces is given later in this section.
- Include in your COleIPFrameWnd derived class some functions performing menu merging.
- Modify some member functions in the COleServerItem derived class to call ActiveX functions when a DocObject is encountered.
- Implement registration.

The IOleObject Interface

The IOleObject interface is derived from OLE 2.0. Therefore, it is useless to implement this interface again if your project is already an OLE 2.0 server application. However, because you must change one of this interface's member functions, you must include the complete interface:

```
BEGIN_INTERFACE_PART(OleObject, IOleObject)
        INIT_INTERFACE_PART(CMyDocObjServer, OleObject)
        STDMETHOD(SetClientSite)(IOleClientSite*);
        STDMETHOD(GetClientSite)(IOleClientSite**);
        STDMETHOD(SetHostNames)(LPCOLESTR, LPCOLESTR);
        STDMETHOD(Close)(DWORD);
        STDMETHOD(SetMoniker)(DWORD, IMoniker*);
        STDMETHOD(GetMoniker)(DWORD, DWORD, IMoniker**);
        STDMETHOD(InitFromData)(IDataObject*, BOOL, DWORD);
        STDMETHOD(GetClipboardData)(DWORD, IDataObject**);
        STDMETHOD(DoVerb)(LONG, LPMSG, IOleClientSite*, LONG,
                          HWND, LPCRECT);
        STDMETHOD(EnumVerbs)(IEnumOLEVERB**);
        STDMETHOD(Update)();
        STDMETHOD(IsUpToDate)();
        STDMETHOD(GetUserClassID)(CLSID*);
        STDMETHOD(GetUserType)(DWORD, LPOLESTR*);
        STDMETHOD(SetExtent)(DWORD, LPSIZEL);
        STDMETHOD(GetExtent)(DWORD, LPSIZEL);
        STDMETHOD(Advise)(IAdviseSink*, LPDWORD);
        STDMETHOD(Unadvise)(DWORD);
        STDMETHOD(EnumAdvise)(IEnumSTATDATA**);
        STDMETHOD(GetMiscStatus)(DWORD, LPDWORD);
        STDMETHOD(SetColorScheme)(LPLOGPALETTE);
    END_INTERFACE_PART(OleObject)
```

Fortunately, because you need to replace only one function, you just have to execute the following changes after the interface is implemented:

```
STDMETHODIMP CMyDocObjectServer::XOleObject::QueryInterface

(REFIID iid, LPVOID* ppvObj)

{

    METHOD_PROLOGUE_EX(CMyDocObjectServer, OleObject)

    return pThis->ExternalQueryInterface(&iid, ppvObj);

}
```

The only member function you must override is SetClientSite(). This member function is given an IOleClientSite interface (which is an OLE 2.0 interface) belonging to a container's document site. The function must perform the following actions:

- Perform standard SetClientSite() processing.
- Release any document site already associated.
- Check whether the document site passed in the argument supports the IOleDocument interface (that is, whether it is a DocObject or just an OLE 2.0 document).

The following source code is an example of SetClientSite() implementation:

```
STDMETHODIMP CMyDocObjectServer::XOleObject::SetClientSite

(IOleClientSite *pDocumentSite)

{

    METHOD_PROLOGUE_EX(CMyDocObjectServer, OleObject)

    HRESULT result = NOERROR;

    // Perform standard SetClientSite processing.

    // m_xOleObject is supposed to point to the ...

    result = m_xOleObject.SetClientSite(pDocumentSite);

    if (result != S_OK)

        return result;

    // If a document site pointer is already associated,

    // release it.

    // m_pDocSite is a CMyDocObjectServer IOleClientSite*

    // variable which points to the associated document site.

    if (m_pDocSite != NULL)

    {
```

```
        m_pDocSite->Release();
        m_pDocSite = NULL;
    }
    // If a document site pointer (i.e. pOleClientSite)
    // is given in argument
    if (pDocumentSite != NULL)
        // Check if it supports the IOleDocumentSite interface.
        // If yes, stores this interface in m_pDocSite.
        result = pDocumentSite->QueryInterface
                 (IID_IOleDocumentSite, (LPVOID*)&m_pDocSite);
    // Return the result of the test.
    return result;
}
```

The IOleDocument Interface

The IOleDocument interface is the front end of the DocObject server application. It is through this interface that the DocObject container demands the creation and enumeration of views.

```
BEGIN_INTERFACE_PART(OleDocument, IOleDocument)
        INIT_INTERFACE_PART(CMyDocObjectServer, OleDocument)
        STDMETHOD(CreateView)(IOleInPlaceSite*, Istream*, DWORD,
                              IOleDocumentView**);
        STDMETHOD(GetDocMiscStatus)(DWORD*);
        STDMETHOD(EnumViews)(IEnumOleDocumentViews**,
                             IOleDocumentView**);
    END_INTERFACE_PART(OleDocument)
```

Following is a list of member functions used in the IOleDocument interface. Each list entry contains a brief description, a syntax statement, and a table containing argument types and returned values.

IOleDocument::CreateView()

The IOleDocument::CreateView() member function asks the DocObject to create a new view. This function returns the created view's IOleDocumentView interface pointer through the last argument, and can take an IStream* parameter for initialization (an IStream is a Win32 interface that allows reading and writing data to stream objects).

```
STDMETHOD(CreateView)(IOleInPlaceSite*, Istream*, DWORD,
                      IOleDocumentView**);
```

Argument Type Description

IOleInPlaceSite* Container's IOleInPlaceSite interface, which must be associated with the new view. If NULL, the view must be associated with a container's view site by calling IOleDocumentView::SetInPlaceSite().

IStream* Stream from which the view should initialize itself. If NULL, the created view maintains a standard initial state.

DWORD Must be zero.

IOleDocumentView** Where to send the new view's IOleDocumentView* interface pointer.

Returned Value Meaning

S_OK The view was created successfully.

E_POINTER The IOleDocumentView** argument is NULL.

E_OUTOFMEMORY No more memory to create the view.

E_UNEXPECTED Unknown error.

E_FAIL This DocObject can only create a single view, which already exists.

IOleDocument::GetDocMiscStatus()

The IOleDocument::GetDocMiscStatus() member function returns the document status through the argument, which is coded as follows:

```
typedef enum
{
    //Object supports multiple views
    DOCMISC_CANCREATEMULTIPLEVIEWS   = 1,
    //IOleDocumentView::SetRectComplex is supported
    DOCMISC_SUPPORTCOMPLEXRECTANGLES = 2,
    //IOleDocumentView::Open is not supported
```

```
    DOCMISC_CANTOPENEDIT            = 4,

    //Object does not support file read/write

    DOCMISC_NOFILESUPPORT           = 8

} DOCMISC;
```

Objects that cannot be edited or can just been in-place activated must have a CANTOPENEDIT status. Objects that only support IPersistStorage as a storage mechanism must have a DOCMISC_NOFILESUPPORT status.

```
STDMETHOD(GetDocMiscStatus)(DWORD*);
```

Argument Type Description

DWORD* Where to store the document status.

Returned Value Meaning

S_OK The status was sent successfully.

E_POINTER The DWORD* argument is NULL.

IOleDocument::EnumViews()

The IOleDocument::EnumViews() member function creates an enumerator that supports the IEnumOleDocumentViews interface. This interface is sent to either the object pointed to by the first argument if the DocObject supports multiple views, or to the object pointed to by the second argument if the DocObject supports only a single view.

```
STDMETHOD(EnumViews)(IEnumOleDocumentViews**, IOleDocumentView**);
```

Argument Type Description

IenumOleDocumentViews** Where to send the enumerator object.

IoleDocumentView** Where to send the enumerator object for a single view.

Returned Value Meaning

S_OK The object was successfully sent.

E_POINTER The appropriate argument is NULL.

E_OUTOFMEMORY No more memory to create the enumerator.

The IEnumOleDocumentViews Interface

Use the IEnumOleDocumentViews interface to control an enumerator. An *enumerator* is an object that provides an IEnumOleDocumentViews interface, and that browses the existing views, returning their IOleDocumentView interfaces. An enumerator is created by calling the IOleDocument::EnumViews() member function.

```
BEGIN_INTERFACE_PART(EnumOleDocumentViews, IEnumOleDocumentViews)
    INIT_INTERFACE_PART(CMyDocObjectServer, EnumOleDocumentViews)
    STDMETHOD(Next)(ULONG, IOleDocumentView**, ULONG*);
    STDMETHOD(Skip)(ULONG);
    STDMETHOD(Reset)(void);
    STDMETHOD(Clone)(IEnumOleDocumentViews**);
END_INTERFACE_PART(EnumOleDocumentView)
```

Following is a list of member functions used in the IEnumOleDocumentViews interface. Each list entry contains a brief description, a syntax statement, and a table containing argument types and returned values.

IEnumOleDocumentViews::Next()

The IEnumOleDocumentViews::Next() member function is given an IOleDocumentView* array (the second argument) in which it puts, at most, a given number (the first argument) of IOleDocumentView* for each existing view. It will also return in the third argument the actual number of views copied.

```
STDMETHOD(Next)(ULONG, IOleDocumentView**, ULONG*);
```

Argument Type Description

ULONG Number of IOleDocumentView* values to put in the array pointed to by the second argument.

IOleDocumentView** Array where view interfaces are stored.

ULONG* Where to send the actual number of document views enumerated. If NULL, the first argument must be 1.

Returned Value Meaning

S_OK The requested number of views has been copied.

S_FALSE There were fewer copied views than requested because there were not enough existing views.

E_POINTER The IOleDocumentView* array is NULL.

E_INVALIDARG The values of the first and last arguments don't match.

E_UNEXPECTED Unknown error.

E_OUTOFMEMORY No more memory to create the enumerator.

IEnumOleDocumentViews::Skip()

The IEnumOleDocumentViews::Skip() member function asks the enumerator to skip a certain number of views. Those views

won't be enumerated again unless IEnumOleDocumentViews::Reset() is called.

```
STDMETHOD(Skip)(ULONG);
```

Argument Type Description

ULONG Number of views to skip.

Returned Value Meaning

S_OK The requested number of views has been skipped.

S_FALSE There were fewer skipped views than requested because there were not enough views left. The enumerator is now positioned at the end of the list.

E_INVALIDARG Invalid argument value.

E_UNEXPECTED Unknown error.

IEnumOleDocumentViews::Reset()

The IEnumOleDocumentViews::Reset() member function asks the enumerator to position itself at the beginning of the list of existing views. The enumerator can then browse all views again.

```
STDMETHOD(Reset)(void);
```

Returned Value Meaning

S_OK The operation succeeded.

S_FALSE The operation failed.

E_UNEXPECTED Unknown error.

IEnumOleDocumentViews::Clone()

The IEnumOleDocumentViews::Clone() member function directs the enumerator to clone itself. The clone is considered to have the same status as the original enumerator, so is has the same position in the views list. The clone's interface is returned through the function argument.

```
STDMETHOD(Clone)(IEnumOleDocumentViews**);
```

Argument Type Description

IEnumOleDocumentViews** Where to send the newly created enumerator's interface.

Returned Value Meaning

S_OK The enumerator was successfully cloned.

E_NOTIMPL Cloning is not supported.

E_POINTER Invalid argument.

E_UNEXPECTED Unknown error.

E_OUTOFMEMORY Not enough memory to create the enumerator.

The IOleDocumentView Interface

The IOleDocumentView interface allows a DocObject container to manage a document view. It is through this interface that the container's view site can set view characteristics, associate a view site with a view, and so on (in order to work properly, each server application's view must be associated with a container's view site).

```
BEGIN_INTERFACE_PART(OleDocumentView, IOleDocumentView

      INIT_INTERFACE_PART(CMyDocObjectServer, OleDocumentView

      STDMETHOD(SetInPlaceSite)(IOleInplaceSite*);

      STDMETHOD(GetInPlaceSite)(IOleInPlaceSite**);

      STDMETHOD(GetDocument)(IUnknown**);

      STDMETHOD(SetRect)(LPRECT);

      STDMETHOD(GetRect)(LPRECT);

      STDMETHOD(SetRectComplex)(LPRECT, LPRECT, LPRECT, LPRECT);

      STDMETHOD(Show)(BOOL);

      STDMETHOD(UIActivate)(BOOL);

      STDMETHOD(Open)(void);

      STDMETHOD(CloseView)(DWORD);

      STDMETHOD(SaveViewState)(IStream*);

      STDMETHOD(ApplyViewState)(IStream*);

      STDMETHOD(Clone)(IOleInPlaceSite*, IOleDocumentView**);

   END_INTERFACE_PART(OleDocumentView)
```

Following is a list of member functions used in the IOleDocumentView interface. Each list entry contains a brief description, a syntax statement, and a table containing argument types and returned values.

IOleDocumentView::SetInPlaceSite()

The IOleDocumentView::SetInPlaceSite() member function associates a container's view site passed in the argument with this view. If the view is already associated with a view site, it must first disassociate from this view site. When a view is associated with (or *holds*) a view site, it only releases it with SetInPlaceSite(NULL) or CloseView(), but NOT with the normal in-place deactivation as do regular OLE documents.

```
STDMETHOD(SetInPlaceSite)(IOleInplaceSite*);
```

Argument Type Description

IOleInPlaceSite* Interface of the site to be associated. If NULL, the view loses any view-site association.

Returned Value Meaning

S_OK Association or disassociation succeeded.

E_FAIL An error occurred.

IOleDocumentView::GetInPlaceSite()

The IOleDocumentView::GetInPlaceSite() member function returns the view site associated with the view through the function argument. This member function returns NULL if the view isn't associated.

```
STDMETHOD(GetInPlaceSite)(IOleInPlaceSite**);
```

Argument Type Description

IOleInPlaceSite** Where to send the view site's interface that is associated with the specific view.

Returned Value Meaning

S_OK The view site was successfully sent.

E_FAIL An error occurred.

IOleDocumentView::GetDocument()

The IOleDocumentView::GetDocument() member function gets the DocObject that owns the view (that is, the DocObject that is shown by the view). This function actually sends a pointer to its IUnknown interface through the function argument.

```
STDMETHOD(GetDocument)(IUnknown**);
```

Argument Type Description

IUnknown** Where to send the IUnknown* interface pointer of the DocObject that owns the view.

Returned Value Meaning

S_OK The interface was successfully sent.

IOleDocumentView::SetRect()

The IOleDocumentView::SetRect() member function sets a new size for the view. The rectangle passed as an argument is in the client's coordinates (that is, the caller's relative coordinates).

```
STDMETHOD(SetRect)(LPRECT);
```

Argument Type Description

LPRECT New size of the view.

Returned Value Meaning

S_OK The view was resized.

E_FAIL An error occurred.

IOleDocumentView::GetRectangle()

The IOleDocumentView::GetRectangle() member function obtains the current view size in client coordinates and returns it through the function argument.

```
STDMETHOD(GetRect)(LPRECT);
```

Argument Type Description

LPRECT Where to store the current view coordinates.

Returned Value Meaning

S_OK The view coordinates were successfully retrieved.

E_UNEXPECTED An error occurred.

IOleDocumentView::SetRectComplex()

The IOleDocumentView::SetRectComplex() member function sets the size of several items: the view port, the scroll bars and the size box. All coordinates passed as arguments must appear as client coordinates.

```
STDMETHOD(SetRectComplex)(LPRECT, LPRECT, LPRECT, LPRECT);
```

Argument Type Description

LPRECT Client Coordinates of the view port.

LPRECT Client coordinates of the horizontal scroll bar.

LPRECT Client coordinates of the vertical scroll bar.

LPRECT Client coordinates of the size box.

Returned Value Meaning

S_OK The view was successfully resized.

E_NOTIMPL Function not supported be the DocObject.

E_FAIL An error occurred.

IOleDocumentView::Show()

The IOleDocumentView::Show() member function asks the view to either show or hide itself.

```
STDMETHOD(Show)(BOOL);
```

Argument Type Description

BOOL If TRUE, the view must be shown. If FALSE the view must be hidden.

Returned Value Meaning

S_OK The view has successfully shown or hidden itself.

E_OUTOFMEMORY Not enough memory.

E_FAIL The operation failed.

E_UNEXPECTED An error occurred.

IOleDocumentView::UIActivate()

The IOleDocumentView::UIActiviate() member function asks the view to either activate or deactivate its user interface elements, such as its menu bar, its toolbars, and so on.

```
STDMETHOD(UIActivate)(BOOL);
```

Argument Type Description

BOOL If TRUE, the view must activated its user interface. If FALSE it must deactivate it.

Returned Value Meaning

S_OK The operation succeeded.

E_OUTOFMEMORY Not enough memory.

E_FAIL The operation failed.

E_UNEXPECTED An error occurred.

IOleDocumentView::Open()

The IOleDocumentView::Open() member function creates a new window that displays the view.

```
STDMETHOD(Open)(void);
```

Returned Value Meaning

S_OK The view successfully created another window.

E_OUTOFMEMORY Not enough memory to create a window.

E_NOTIMPL Function not supported.

E_FAIL The operation failed.

E_UNEXPECTED An error occurred.

IOleDocumentView::CloseView()

The IOleDocumentView::CloseView() member function closes the view, releasing any view site associated with it. The container must call this member function before deleting the view.

```
STDMETHOD(CloseView)(DWORD);
```

Argument Type Description

DWORD Must be zero.

Returned Value Meaning

S_OK The view coordinates were successfully closed.

Closing a view is not supposed to fail, which is why S_OK is the only listed return value.

IOleDocumentView::SaveViewState()

The IOleDocumentView::SaveViewState() member function asks the view to save its state in an IStream passed in an argument. This function is typically used before closing a view; the stream keeps its state, allowing the state to be restored later via IOleDocumentView::ApplyViewState().

```
STDMETHOD(SaveViewState)(IStream*);
```

Argument Type Description

IStream* Where the state is saved.

Returned Value Meaning

S_OK The view state was successfully saved.

E_POINTER Invalid argument.

E_NOTIMPL Function not supported.

IOleDocumentView::ApplyViewState()

The IOleDocumentView::ApplyViewState() member function opposes IOleDocumentView::SaveViewState(). IOleDocumentView::ApplyViewState() restores a state saved in the stream passed as an argument.

```
STDMETHOD(ApplyViewState)(IStream*);
```

Argument Type Description

IStream* The location from which the state is loaded.

Returned Value Meaning

S_OK The view state was successfully loaded.

E_POINTER Invalid argument.

E_NOTIMPL Function not supported.

IOleDocumentView::Clone()

The IOleDocumentView::Clone() member function creates a copy of its view object, with the same state (though associated with a different view site). The first argument is the container's view site for the clone, whose IOleDocumentView interface is returned through the last argument.

```
STDMETHOD(Clone)(IOleInPlaceSite*, IOleDocumentView**);
```

Argument Type Description

IOleInPlaceSite* In-place site to be associated with the clone.

IOleDocumentView** Where to store the pointer of the new view.

Returned Value Meaning

S_OK The view was successfully cloned.

E_POINTER Invalid argument.

E_FAIL The DocObject supports only a single view.

Modify the COleIPFrameWnd Derived Class.

A DocObject server application must add functions to its COleIPFrameWnd derived class. Indeed, this class must implement the server application's part of menu merging. This implementation can be divided into two parts: the creation of the menu and its destruction.

For more information about how DocObject menu merging works, see section later in this chapter titled "Help Menu Merging."

Creating a Merged Menu'

Menus are merged whenever the server application is activated (when OleDocumentView::UIActivate(BOOL) is passed a TRUE argument).

The COleIPFrameWnd class provides four useful variables for menu merging:

```
LPOLEINPLACEFRAME m_lpFrame;

    HMENU m_hSharedMenu;

    OLEMENUGROUPWIDTHS m_menuWidths;

    HOLEMENU m_hOleMenu;
```

m_lpFrame points to the container's IOleInPlaceFrame interface. m_hSharedMenu contains the merged menu bar; m_menuWidths contains information about which menus are inserted by the container or by the server application. Finally, m_hOleMenu contains the menu descriptor.

The server application must set the m_hSharedMenu and m_menuWidths member variables, which are then sent to the container. The m_menuWidths array must be initialized with zeros, and the m_hSharedMenu variable must represent a blank menu bar by calling

```
    m_hSharedMenu =::CreateMenu();
```

The server application then passes the blank menu bar and the OLEMENUGROUPWIDTHS array to the container by calling

```
    m_lpFrame->InsertMenus(m_hSharedMenu, &m_menuWidths);
```

After the container has inserted its menus, the server application inserts desired menus by modifying m_hSharedMenu and m_menuWidths. This procedure is the standard OLE 2.0 menu merging, except that if the container inserted a Help menu, the server application must:

- Set m_menuWidths[5] to 0.
- Increment m_menuWidths[5] by 1 (so that OLE recognizes a Help menu merging).
- Add its own Help menu in the container's menu as a sub-menu.

The server application must then sends the resulting menu to OLE and get back the menu's descriptor:

```
m_hOleMenu =::OleCreateMenuDescriptor(m_hSharedMenu,
                                      &m_menuWidths);
```

Last but not least, the server application must return m_hOleMenu to the container.

Destroying the Merged Menu'

When the DocObject is destroyed, the merged menu must also be destroyed. The container removes its menus from the menu bar and sends the resulting menu to the server application, which must also removes the menus it inserted.

The server application removes the menus (standard OLE procedure), and then destroys m_hSharedMenu by calling

```
::DestroyMenu(m_hSharedMenu);
```

Modify the COleServerItem Derived Class

The COleServerItem derived class must be slightly modified to be able to distinguish a DocObject client from a regular OLE 2.0 client. Indeed, if the IOleObject::DoVerb() is passed an OLEIVERB_SHOW, an OLEIVERB_OPEN, or an OLEIVERB_UIACTIVATE order, the DocObject server application must call the IOleDocumentSite::ActivateMe() interface member function. If the IOleObject::DoVerb() is passed an OLEIVERB_HIDE order, it must raise an E_INVALIDARG error.

To configure the COleServerItem derived class to distinguish a DocObject client from an OLE 2.0 client, implement the following changes:

- The OnShow() and OnOpen() member functions must check whether the client is a DocObject (by determining whether the client has an associated document site, for example). If the client is a DocObject, the OnShow() and OnOpen() member functions must call IOleDocumentSite::ActivateMe(). If the client is an OLE 2.0 client, these member functions simply call the regular COleServerItem::OnShow() or COleServerItem::OnOpen() member functions.
- The OnHide() member function must perform the same check. If the client is a DocObject container, the OnHide() member function must raise an error by calling AfxThrowOleException(). Otherwise, the OnHide() member function simply calls the regular COleServerItem::OnHide() member function.

Add MFC Registry

The server application must add its keys to the operating system registry (that is, the Windows 95 or Windows NT registry), which registers the server application with the system as a DocObject server application. For that purpose, the server application must add:

```
HKEY_CLASSES_ROOT\My.DocObject.Server.1\DocObject = 0

HKEY_CLASSES_ROOT\CLSID\<The server's CLSID> = My Document Object

HKEY_CLASSES_ROOT\CLSID\<The server's CLSID>\DocObject = 0
```

```
HKEY_CLASSES_ROOT\CLSID\<The server's CLSID>\DefaultExtention = .my, My DocObject
(*.my)

HKEY_CLASSES_ROOT\CLSID\<The server's CLSID>\Printable
```

The last key must be added only if the server application supports the IPrint interface.

The DocObject server application can modify status bits from the MiscStatus key in the registry, which gives the server application the option of not supporting certain troublesome OLE 2.0 features. OLEMISC_CANTLINKINSIDE prevents linking to embedded DocObjects. OLEMISC_ICONICONLY forces the DocObjects to appear only as an icon when copied into containers.

Miscellaneous Stuff

Here are a few more things to remember when implementing a DocObject server application:

- When the DocObject is activating, the server application should ignore calls such as IOleObject::SetExtent(), and should not use the IOleInPlaceSite::OnPosRectChange() orIOleClientSite::ShowObject() member functions. The server application should draw scroll bars within the view rectangle using the IOleDocumentView::SetRect() or IOleDocumentView::SetRectComplex() interface member functions.
- Only use IStorage and IStream for storage purposes. Also, the storage format must always be the same, whether the DocObject manages its own storage file or uses a storage medium provided by the container. A DocObject container can create a file from the server application's objects storage.
- Although a DocObject can behave like an OLE 2.0 document, a DocObject can decide not to support some OLE 2.0 features. Disabling Embed Source or Embedded Object formats in data exchange operations prevents the DocObject from being pasted within compound document containers.

DocObject Container Implementation

The implementation of a DocObject container requires the creation of DocObject-specific interfaces and several other functions. A list of the necessary interfaces follows:

- IOleDocumentSite interface—interface of the document site.
- IOleInPlaceSite interface (OLE 2.0 interface, optional)—interface of the view site.
- IOleCommandTarget interface (optional)—command target. See the section found later in this chapter titled "Command Target."
- IContinueCallBack interface (optional)—programmatic printing. See the section found later in this chapter titled "Programmatic Printing."

DocObject Initialization

When a container creates a DocObject, it can create a brand new DocObject with a standard initial state, or the container can create a DocObject from a file or from data (through a cut & paste operation, for example). The full initialization of the DocObject depends on how the DocObject was created. However, initialization will at least require:

- IPersistStorage::InitNew() or IPersistStorage::Load() to create the DocObject.
- IOleObject::SetClientSite() to pass the container's address to the DocObject.

- IOleObject::Advise() if the IAdviseSink interface is supported.

The IOleDocumentSite Interface

The IOleDocumentSite interface is the only ActiveX interface required to create a container site (all other ActiveX interfaces are optional). The goal of this interface is to allow a DocObject to ask the container to activate itself.

```
BEGIN_INTERFACE_PART(OleDocumentSite, IOleDocumentSite)

    INIT_INTERFACE_PART(CMyDocObjectContainer, OleDocumentSite)

    STDMETHOD(ActivateMe)(IOleDocumentView*);

END_INTERFACE_PART(OleDocumentSite)
```

Following is a list of member functions used in the IOleDocumentSite interface. Each list entry contains a brief description, a syntax statement, and a table containing argument types and returned values.

IOleDocumentSite::ActivateMe()

The IOleDocumentSite::ActivateMe() member function asks a DocObject container to activate itself. This member function is called by the DocObject server application in the server's IOleObject::DoVerb() interface member function. It is passed the interface of the server application's view to activate. Following is a syntax statement and a table containing argument types and returned values.

```
STDMETHOD(ActivateMe)(IOleDocumentView*);
```

Argument Type Description

IOleDocumentView* The view to be activated.

Returned Value Meaning

S_OK The DocObject container successfully activated the view.

E_OUTOFMEMORY Not enough memory to create the view.

E_FAIL An error occurred in the process.

The IOleDocumentSite::ActivateMe() member function performs the following tasks:

- If IOleDocumentSite::ActivateMe()is given an non-NULL IOleDocumentView* argument, IOleDocumentView::SetInplaceSite() and IOleDocumentView::AddRef() must be called.
- If the passed argument is given a NULL pointer, the function must get the DocObject server application's IOleDocument interface by calling IOleObject::QueryInterface(). Through this interface, ActivateMe() then calls IOleDocument::CreateView(), thus creating a view and retrieving its IOleDocumentView interface.
- |lb] The function must activate its newly acquired IOleDocumentView interface by calling IOleDocumentView::UIActivate(TRUE).
- ActivateMe() then calls IOleDocumentView::SetRect() or IOleDocumentView::SetRectComplex() tell the view what space to occupy.

- Finally, the function calls IOleDocumentView::Show(TRUE) to make the DocObject display itself.

Design the Data Storage Structure Based on the User's Document Embedding aspect.

A DocObject container can manage multiple documents in a single data storage structure. This data storage structure can be an OLE compound file, or it can be divided into several files (one file per document). No matter which method you choose, always design your data storage structure to correspond with the user's views on document embedding.

For example, Office Binder uses an OLE compound file to store all documents. So it is normal to have every DocObject of a Binder catalog (the *Binder catalog* being the native Office Binder format) in one unique compound file.

On the other hand, take, for example, a container dealing with documents spread across the Internet. This time, DocObjects are not seen as being contained in a binder. Therefore, in this case, using several files would be more effective than using a compound file.

Closing a DocObject

When a DocObject is closed, the container should check whether its data was saved. It must call IOleClientSite::SaveObject() if needed, and then place IOleInplaceObject::InPlaceDeactivate(), IOleObject::Close() and IUnknown::Realease() calls to every interface it is referencing.

Optional OLE Functions and Necessary OLE Functions

The container can simply return the message ENOTIMPL instead of implementing these optional OLE functions:

```
IOleClientSite::GetMoniker()

IOleClientSite::GetContainer()

IOleClientSite::RequestNewObjectLayout()

IOleClientSite::OnShowWindow()

IOleClientSite::ShowObject()

IOleInPlaceSite::OnPosRectChange()

IOleInPlaceSite::Scroll()

IOleInPlaceSite::ContextSensitiveHelp()
```

Every AdviseSink interface member function is optional except AdviseSink::OnClose(), which must be implemented.

All other member functions of IOleClientSite, IOleInPlaceSite and IOleInPlaceFrame need some implementation. Moreover, if the container supports OLE-only documents, all normal OLE functions must be implemented.

Miscellaneous Stuff

There are still a few things you should know in order to implement a DocObject container:

- Call IOleInPlaceActiveObject::ResizeBorder() when the container frame size has changed. This way, the DocObject can resize its tool bars.
- Call IOleDocumentView::SetRect() when any type of window containing the DocObject is resized (frame windows, MDI window).
- If the container has a toolbar, you must implement the member function IOleInPlaceFrame::SetStatusText().
- IOleInPlaceActiveObject::TranslateAccelerator() must be called within the container's message pump.
- Call IOleInplaceActiveObject::ContextSensitiveHelp() to detect the F1 key (to display online help) or the ESC key (to close online help).
- |lb] Have the frame window handle WM_SETFOCUS by setting focus to the window returned by IOleInPlaceActiveObject::GetWindow().

Programmatic Printing

Programmatic printing is a new DocObject feature that allows the DocObject container to manage the printing of embedded DocObjects. Programmatic printing provides an interface for printing, for getting the status of print jobs, for setting the pages to be printed, and even for providing a callback mechanism with which the user can halt the printing process before its end. In other words, the container can ask the embedded object to print itself. However, the container still has the control in case the user decides to abort the printing process.

Programmatic printing is composed of two interfaces, IPrint and IContinueCallBack. The IPrint interface controls the printing process itself; it sets the initial page number, decides which pages will be printed, and retrieves printing status information (such as the number of pages that were actually printed and the last printed page number). This interface must be implemented in the DocObject server application. The IContinueCallback interface provides a way for the user to abort the printing process. Actually, IContinueCallback can be used for any lengthy operation. A background process regularly checks whether the user has decided to abort the printing process. If so, it calls an IContinueCallback member function, which asks the user whether he wants to abort printing or not. This interface must be implemented in the DocObject container.

The IPrint Interface

The IPrint interface implements the document printing management; it sets the pages to print, prints them, and retrieves print-related information.

```
BEGIN_INTERFACE_PART(Print, IPrint)

      INIT_INTERFACE_PART(CMyDocObjectServer, Print)

      STDMETHOD(SetInitialPageNum)(LONG);

      STDMETHOD(GetPageInfo)(LONG*, LONG*);

      STDMETHOD(Print)(DWORD, DVTARGETDEVICE**, PAGESET**,
```

```
                    STGMEDIUM*, IContinueCallBack*, LONG,
                    LONG*, LONG*);
END_INTERFACE_PART(Print)
```

Before describing the interface member functions, let's take a look at some structures used by these functions. Each of the following entries contains a brief description and a table; some entries contain a syntax statement.

The PAGERANGE Structure

As its names implies, the PAGERANGE structure contains a range of pages.

Member Type Description

nFrontPage LONG The first page to print.

nToPage LONG The last page to print.

The PAGESET Structure

The PAGESET structure identifies a set of page ranges, and optionally indicates information such as odd or even pages.

Member Type Description

cbStruct ULONG Number of bytes in this PAGESET structure. Must be a multiple of 4.

FOddPages BOOL If TRUE, print only odd pages.

fEvenPages BOOL If TRUE, print only even pages.

cPageRange ULONG Number of page ranges in rgPages.

rgPages PAGERANGE* Contains the set of page ranges to be printed. The range must be sorted in increasing order and must not overlap.

The PRINTFLAG Enumeration

The PRINTFLAG enumeration contains miscellaneous values.

Value Member Description

PRINTFLAG_MAYBOTHERUSER Indicates that the printing process can interact with the user.

PRINTFLAG_PROMPTUSER Indicates the normal print dialog box must be used.

PRINTFLAG_USERMAYCHANGEPRINTER The user may change the printer.

PRINTFLAG_RECOMPOSETODEVICE The document must recompose itself to the printer.

PRINTFLAG_DONTACTUALLYPRINT Only simulate printing.

PRINTFLAG_PRINTTOFILE Print to a file.

IPrint::SetInitialPageNum()

The IPrint::SetInitialPageNum() member function the first page to be printed.

```
STDMETHOD(SetInitialPageNum)(LONG);
```

Argument Type Description

LONG First page number.

Returned Value Meaning

S_OK The first page was set as desired.

E_FAIL The first page could not set as desired.

E_UNEXPECTED An error occurred.

IPrint::GetPageInfo()

The IPrint::GetPageInfo() member function returns information about pages through its two arguments.

```
STDMETHOD(GetPageInfo)(LONG*, LONG*);
```

Argument Type Description

LONG* Where to copy the page number of the first page. A NULL value means the caller doesn't need this number.

LONG* Where to copy the total number of pages in the document. A NULL value means the caller doesn't need this number.

Returned Value Meaning

S_OK The operation succeeded.

E_UNEXPECTED An error occurred.

IPrint::Print()

The IPrint::Print() member function asks the object to print itself. The first six arguments are parameters for the printing process, like the pages to be printed, the printing process's target device and so on. This function returns printing information through its last two arguments.

```
STDMETHOD(Print)(DWORD, DVTARGETDEVICE**, PAGESET**, STGMEDIUM*,
```

```
                IContinueCallBack*, LONG, LONG*, LONG*);
```

Argument Type Description

DWORD A PRINTFLAG enumeration.

DVTARGETDEVICE** The target device.

PAGESET** Pages to be printed.

STGMEDIUM** Document-specific printing functions.

IContinueCallback* A callback interface allowing the user to cancel printing.

LONG The starting page number.

LONG* Where to copy the actual number of successfully printed pages.

LONG* Where to copy the last page number.

Returned Value Meaning

S_OK The pages were successfully printed.

PRINT_E_CANCELLED The operation was canceled.

PRINT_E_NOSUCHPAGE A page that was to be printed does not exist.

E_UNEXPECTED An error occurred.

The IContinueCallback Interface

The IContinueCallBack interface is manages a generic callback procedure, allowing the user interrupt running (lengthy) processes.

```
BEGIN_INTERFACE_PART(ContinueCallback, IContinueCallback)
      INIT_INTERFACE_PART(CMyDocObject, ContinueCallback)
      STDMETHOD(FContinue)(void);
      STDMETHOD(FContinuePrinting)(LONG, LONG, wchar_t*);
END_INTERFACE_PART(ContinueCallback)
```

Let's take a look at some member functions of this interface. Each of the following entries contains a brief description a table, a syntax statement.

IContinueCallback::FContinue()

The IContinueCallback::FContinue() member function administers the continuation (or discontinuation) of a given operation.

```
STDMETHOD(FContinue)(void);
```

Returned Value Meaning

S_OK Continue the operation.

E_FALSE Cancel the operation as soon as possible.

IContinueCallback::FContinuePrinting()

The IContinueCallback::FContinuePrinting() member function administers the continuation (or discontinuation) of a given printing process.

```
STDMETHOD(FContinuePrinting)(LONG, LONG, wchar_t*);
```

Argument Type Description

LONG Actual number of printed pages.

LONG The page number of the page being printed.

wchar_t* Status message.

Returned Value Meaning

S_OK Continue printing.

S_FALSE Cancel printing as soon as possible.

E_UNEXPECTED An error occurred.

Command Target

Command target is a mechanism that allows the container to query and execute one or more commands from the server application, and vice-versa. In other words, a DocObject's container and server application can ask each other whether or not the requested command is supported. If so, they can request its execution.

Commands all belong to a group. An example of group is the standard command list, which regroups all standard commands from Microsoft Office applications, such as File Open, File Close, and so on. However, anybody can create a group containing its own commands.

Command target can be useful in the following cases:

- The container might want to send commands to the server application, such as printing commands and page-setup commands. The container can determine whether the server application supports these functions. If not, the container can disable these features in its own menu.
- The DocObject can have in its own toolbar icons referencing container functions, such as New, Save, and so on. Once again, the DocObject can determine whether the container supports these commands; if not, the DocObject can remove them from its own toolbar. If the command is supported by the container, the DocObject can then forward the command to the container any time a user pushes the appropriate DocObject toolbar icon.

Use the IOleCommandTarget to load the command target interface, which can be implemented in the DocObject's container, the server application, or both (depending on what comment target functionality is required).

The IOleCommandTarget Interface

The IOleCommandTarget interface enables to ask if a command is available and to execute it. The commands are referenced by an integer, within a group of commands. The group itself is referenced by an integer. Through this interface, one can ask whether a given command is supported and/or execute it.

```
BEGIN_INTERFACE_PART(OleCommandTarget, IOleCommandTarget)

    INIT_INTERFACE_PART(CMyDocObject, OleCommandTarget)

    STDMETHOD(QueryStatus)(const GUID*, ULONG, OLECMD*, OLECMDTEXT*);

    STDMETHOD(Exec)(const GUID*, DWORD, DWORD,

                    VARIANTARG*, VARIANTARG*);

END_INTERFACE_PART(OleCommandTarget)
```

This interface uses some structures and enumerations, which will be explained before its member functions. The following entries contain a brief description and a table; member function entries also contain a syntax statement.

The OLECMDF Enumeration

The OLECMDF enumeration contains values describing the status of a given command.

Value Description

OLECMDF_SUPPORTED The command is supported.

OLECMDF_ENABLED The command is enabled.

OLECMDF_LATCHED The command is an on/off switch, and is currently on.

OLECMDF_NINCHED The command is an on/off switch, but the state is indeterminate.

The OLECMD Structure

The OLECMD structure contains a command identifier and its status.

Member Type Description

cmdID ULONG Command identifier.

cmdf DWORD OLECMDF enumeration.

The OLECMDTEXTF Enumeration

This enumeration contains flags that describe that information to return after the command was executed.

Argument Type Description

OLECMDTEXTF_NONE No extra information is requested.

OLECMDTEXTF_NAME Return the command name.

OLECMDTEXTF_STATUS Return a status string.

The OLECMDTEXT Structure

The OLECMDTEXT structure is used to return text name or status string.

Member Type Description

cmdtextf DWORD OLECMDTEXTF enumeration.

cwActual ULONG Where to store the number of characters actually written into rgwz.

cwBuf ULONG Size of the string buffer.

rgwz wchar_t Array that will receive the output string.

The OLECMDEXECOPT Enumeration

The OLECMDEXECOPT enumeration contains flags that describe what information should be returned after the command was executed.

Argument Type Description

OLECMDTEXTF_NONE No extra information is requested.

OLECMDTEXTF_NAME Return the command name.

OLECMDTEXTF_STATUS Return a status string.

The OLECMDID Enumeration

The OLECMDID enumeration is known as the *Standard Command List*. This list contains several of the standard commands defined in Microsoft Office 95. Their group is referenced with a NULL identifier.

Argument Type Description

OLECMDID_OPEN File Open

OLECMDID_NEW File New

OLECMDID_SAVE File Save

OLECMDID_SAVEAS File Save As

OLECMDID_SAVECOPY File Save Copy As

OLECMDID_PRINT File Print

OLECMDID_PRINTPREVIEW File Print Preview

OLECMDID_PAGESETUP File Page Setup

OLECMDID_SPELL Tools Spelling

OLECMDID_PROPERTIES File Properties

OLECMDID_CUT Edit Cut

OLECMDID_COPY Edit Copy

OLECMDID_PASTE Edit Paste

OLECMDID_PASTESPECIAL Edit Paste Special

OLECMDID_UNDO Edit Undo

OLECMDID_REDO Edit Redo

OLECMDID_SELECTALL Edit Select All

OLECMDID_CLEARSELECTION Edit Clear Selection

OLECMDID_ZOOM View Zoom

OLECMDID_GETZOOMRANGE Get Zoom Range.

OLECMDID_UPDATECOMMANDS Tells when one can retrieve the commands status at a convenient time.

OLECMDID_REFRESH Display refresh request.

OLECMDID_STOP Stop all current processing.

OLECMDID_HIDETOOLBARS Hide the tool bars of the object.

OLECMDID_SETPROGRESSMAX Sets the maximum value of the progress indicator owned by the object.

OLECMDID_SETPROGRESSPOS Sets the current value of the progress indicator.

OLECMDID_SETPROGRESSTEXT Sets the text of the progress indicator.

OLECMDID_SETTITLE Sets the window title.

The commands OLECMDID_ZOOM and OLECMDID_GETZOOMRANGE need some arguments, passed through IOleCommandTarget::Exec().

OLECMDID_ZOOM

The OLECMDID ZOOM command manages the zoom. It takes a LONG input argument and returns a LONG output argument. This command can be used for three different purposes:

- To get the current zoom value, send NULL as the input argument, specifying the OLECMDEXECOPT_DONTPROMPTUSER option (third argument of Exec()). The zoom value will be returned in the output argument.
- To set the zoom value, pass the desired value in the input argument, specifying the OLECMDEXECOPT_DONTPROMPTUSER option.
- To let the user choose the zoom value, pass an (optional) initial value as an input argument, specifying the OLECMDEXECOPT_PROMPTUSER option. The zoom dialog box pops up to prompt the user. The output argument

will return the value selected by the user, or OLECMDERR_E_CANCELLED will be returned if the user canceled the operation.

OLECMDID_GETZOOMRANGE

The OLECMDID GETZOOMRANGE command is used to determine the range of valid zoom values. The input argument is NULL, and the execution option is MSOCMDEXECOPT_DONTPROMPTUSER. The command returns its output argument as DWORD. The HIWORD contains the maximum zoom value, whereas the LOWORD contains the minimum zoom value.

IOleCommandTarget::QueryStatus()

The IOleCommandTarget::QueryStatus() member function returns the status of one or more commands. This function is typically called after WM_INITMENU or WM_INITMENUPOPUP messages. Indeed, before displaying its menu bar or pop-up menu, a container might want to ask the server application whether some of its commands are supported so that unsupported ones can be removed from its menus. The first three arguments relate to command description, such as its group and so forth. The third argument is an array containing all commands whose status must be queried. The result is sent through the function's last argument.

```
STDMETHOD(QueryStatus)(const GUID*, ULONG, OLECMD*, OLECMDTEXT*);
```

Argument Type Description

const GUID* Identifier of the command group. A NULL value indicates the standard group.

ULONG Number of commands in the OLECMD array.

OLECMD* Array of OLECMD structures.

OLECMDTEXT* Where to send name and/or status information. A NULL value indicates that the caller doesn't need this information.

Returned Value Meaning

S_OK Continue printing.

E_POINTER The third argument is NULL.

E_UNEXPECTED An unexpected error occurred.

E_FAIL An error occurred.

OLECMDERR_E_UNKNOWNGROUP The group type (first argument) is invalid.

IOleCommandTarget::Exec()

The IOleCommandTarget::Exec() member function executes a given command or displays its help information. Although most commands do not need any argument, the fourth parameter was added for this purpose.

```
STDMETHOD(Exec)(const GUID*, DWORD, DWORD,
                VARIANTARG*, VARIANTARG*);
```

Argument Type Description

const GUID* Identifier of the command group. A NULL value indicates the standard group.

DWORD The command to execute.

DWORD One or more values from OLECMDEXECOPT enumeration.

VARIANTARG* Input arguments. Might be NULL.

VARIANTARG* Output values. Might be NULL.

Returned Value Meaning

S_OK Continue printing.

E_UNEXPECTED An unexpected error occurred

E_FAIL An error occurred.

OLECMDERR_E_UNKNOWNGROUP The group type (first argument) is invalid.

OLECMDERR_E_NOTSUPPORTED The command is invalid.

OLECMDERR_DISABLED The command is currently disabled.

OLECMDERR_NOHELP The caller asked for help, which is not available.

OLECMDERR_CANCELLED The user canceled an operation.

Help Menu Merging

When a container embeds a document using OLE 2.0, the OLE client and server application's menus are merged. Actually, some parts of the menu are managed by the client (such as File and Window), and some parts are managed by the server application (such as Edit, View and other server-application-specific menus). The Help menu is by default managed by the OLE client. However, when the user edits the embedded document, it is replaced by the server application's Help menu.

However, because a DocObject can be embedded without being contained by a native container document, OLE 2.0 menu merging raises a problem: which Help menu should you include? While putting the server application's help menu might seem logical, it raises a problem with DocObject containers (such as Office Binder): help would only be available when the Binder catalog contained no documents.

The solution instituted by DocObjects is to include both the container's and the server application's help information in a unique Help menu composed of two cascade menus: one for the container and one for the server application. This Help menu is described as follows:

```
Help
      Container Help >
      Server Help    >
```

How OLE 2.0 Performs Menu Merging

The DocObject approach to menu merging differs from OLE 2.0's approach. Indeed, with OLE 2.0, the merged menu bar can have 6 groups of menus: File, Edit, Container, Object, Window, and Help. Each of the 6 groups can represent some menus (they can however represent no menu at all. For example a program that doesn't support cut & paste operations doesn't need an Edit menu group). The File, Container and Window groups are managed by the container, whereas the Edit, Object and Help groups are managed by the OLE document.

The mains steps of OLE 2.0 menu merging follow:

1. 1. The OLE server application creates a blank menu bar, along with a six-element OLEMENUGROUPWIDTHS array, and passes both to the client.
2. 2. The OLE client inserts its own menu elements within the menu bar and writes in the array which groups were inserted. It then returns the menu bar and the array to the server application.
3. 3. The server application then inserts its own menus, checking which groups were inserted by the client, and updating the array accordingly. The server application passes the resulting menu and array to the operating system's OLE subsystem, which returns the merged menu descriptor handle.
4. 4. The menu bar and handle are passed to the container. The container displays the menu bar and sends the handle to the OLE subsystem to dispatch the menu messages correctly.

Implementation Differences between DocObjects and OLE 2.0

Implementing menu merging in DocObjects differs slightly from OLE 2.0's menu-merging-implementation process; DocObjects require you to implement a few extras. The following explains those steps not required in OLE 2.0 to implement menu merging in DocObjects:

1. 1. The server application clears the array after its creation.
2. 2. The client inserts a Help menu element as the last item and writes a 1 in the last array entry (that is Array[5]=1), indicating the insertion of a Help group menu element.
3. 3. The server application determines whether the client inserted a Help group menu. If so, the server application inserts its help pop-up menu as a sub-menu of the existing Help pop-up menu (the client Help menu). The server application then sets the last array element to 0, but increments the element before it (Array[4]) by 1. OLE then knows that the Help menu must be merged.
4. 4. The client forwards all menu messages that concern the embedded-document portion of the Help menu to the server application.
5. 5. While the menu is being destroyed, the server application removes all menus and sub-menus that it inserted. The client must do the same job when it receives the menu bar.

How to Create a Basic DocObject Server

Let's look at how to create a basic DocObject server application from a basic Visual C++ 4 AppWizard application. Although Visual C++ 4.2 should include this process in one of its AppWizards, it isn't supported in previous versions of Visual C++ 4. Furthermore, some people might be willing to turn their existing OLE 2.0 server application into a DocObject server application, even if they are working with Visual C++ 4.2.

Implementing a whole DocObject server application with all the DocObject interfaces would take a lot of time. A simpler way to achieve the same effect is to borrow some files from the BINDSCRB sample shipped with Visual C++ 4.0. This sample is a little application with DocObject server application features.

The creation of a DocObject server application will take the following three steps:

- Create an AppWizard OLE server application.
- Modify the AppWizard-generated application to have an operational OLE server application.
- Modify the OLE server application to have a DocObject server application.

The first two steps explain how to create an OLE 2.0 server application. Although creating an OLE 2.0 server application is not the goal of this chapter, some readers might find this information useful. Readers interested in turning their OLE server application into a DocObject-compliant server application can skip the first two steps and jump directly to the last one.

Creating an AppWizard Application

Although none of the three steps are really painful, creating an AppWizard Application is the easiest and fastest step. First, create an MFC AppWizard project and name it DoServer (for a DocObject Server application). You can enter a name other than DoServer if you wish, but I will refer to the MFC AppWizard project as DoServer. Next, in step 3 of the MFC AppWizard dialog box, select the option «[dg]Full-Server[dg]».

The created project will be an OLE server application with no attached document (see Figure 18.7).

Figure 18.7. The AppWizard options.

Modify Your Application to Make It an Operational OLE Server Application

The application created by AppWizard, which is virtually an OLE server application, doesn't display anything. This can be problematic if you want to test it. That is why I add a few lines of source code to make the server application display Hello, ActiveX world. In the following entries, bold code represents the code to be added or modified. Normal text code represents the existing code (created by AppWizard).

First, alter the CDoServerDoc class and add the member functions Draw() and GetExtent(), which display the text and calculate the size of the string, respectively. Also, add a few member variables for text management. In the DoServerDoc.h file, add the following bold lines:

```
class CDoServerDoc: public COleServerDoc

{

...

public:

    void CDoServerDoc::Draw(CDC *pDC) const;

    void CDoServerDoc::GetExtent(CDC *pDC, CSize &size) const;

protected:

    CString m_text;

    LOGFONT m_logfont;
```

```
    COLORREF m_crText;
...
};
```

In the sample on the CD-ROM that accompanies this book, every line added to the AppWizard application contains the following comment:

```
...   // OLE server addition
```

The class must now initialize the font. In the DoServerDoc.cpp file, add the following code:

```
CDoServerDoc::CDoServerDoc()
{
    memset(&m_logfont, 0, sizeof m_logfont);
    m_logfont.lfHeight = -10;
    lstrcpy(m_logfont.lfFaceName, _T("Arial"));
    m_logfont.lfOutPrecision = OUT_TT_PRECIS;
    m_logfont.lfClipPrecision = CLIP_DEFAULT_PRECIS;
    m_logfont.lfQuality = PROOF_QUALITY;
    m_logfont.lfPitchAndFamily = FF_SWISS | VARIABLE_PITCH;
    m_crText = COLOR_WINDOWTEXT+1;
    m_text = "Hello, ActiveX World !";
    ...
};
```

In the DoServerView.cpp file, The CDoServerView::OnDraw() member function must be altered to ask the document to display itself:

```
void CDoServerView::OnDraw(CDC* pDC)
{
    CDoServerDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
```

```
    // TODO: add draw code for native data here

    pDoc->Draw(pDC);

}
```

The CDoServerSrvrItem::OnGetExtent() member function, whose goal is to ask the document to calculate its own size, must be implemented. In the SrvItem.cpp file, add:

```
BOOL CDoServerSrvrItem::OnGetExtent(DVASPECT dwDrawAspect, CSize& rSize)

{

...

    // TODO: replace this arbitrary size

    CClientDC dc(NULL);

    dc.SetMapMode(MM_ANISOTROPIC);

    pDoc->GetExtent(&dc, rSize);

    dc.LPtoHIMETRIC(&rSize);

    return TRUE;

}
```

The CDoServerSrvrItem::OnDraw() member function, whose goal is to prepare the device context for drawing and to ask the document to draw itself, must be implemented. In the SrvItem.cpp file, add:

```
BOOL CDoServerSrvrItem::OnDraw(CDC* pDC, CSize& rSize)

{

    CDoServerDoc* pDoc = GetDocument();

    ASSERT_VALID(pDoc);

    // TODO: set mapping mode and extent

    //  (The extent is usually the same as

    // the size returned from OnGetExtent)

    OnGetExtent(DVASPECT_CONTENT, rSize);

    pDC->SetMapMode(MM_ANISOTROPIC);

    pDC->SetWindowOrg(0,0);

    pDC->SetWindowExt(rSize.cx, rSize.cy);
```

```
    // TODO: add drawing code here.  Optionally, fill in the
    // HIMETRIC extent. All drawing takes place in the metafile
    // device context (pDC).
    pDoc->Draw(pDC);
    return TRUE;
}
```

Last but not least, the Draw() and GetExtent() member functions must be implemented in the DoServerDoc.cpp file:

```
void CDoServerDoc::Draw(CDC *pDC) const
{
    LOGFONT logfont = m_logfont;
    logfont.lfHeight = -::MulDiv(-logfont.lfHeight,
                            pDC->GetDeviceCaps(LOGPIXELSY), 72);
    pDC->SetTextColor(m_crText);
    CFont font;
    if (!font.CreateFontIndirect(&logfont))
    {
        CFont *pOldFont = pDC->SelectObject(&font);
        pDC->TextOut(0, 0, m_text, m_text.GetLength());
        pDC->SelectObject(pOldFont);
    }
    else
        pDC->TextOut(0, 0, m_text, m_text.GetLength());
}
void CDoServerDoc::GetExtent(CDC *pDC, CSize &size) const
{
    LOGFONT logfont = m_logfont;
    logfont.lfHeight = -::MulDiv(-logfont.lfHeight,
```

```
                                pDC->GetDeviceCaps(LOGPIXELSY), 72);
    CFont font;
    if (!font.CreateFontIndirect(&logfont))
    {
        CFont *pOldFont = pDC->SelectObject(&font);
        TEXTMETRIC tm;
        pDC->GetTextMetrics(&tm);
        size.cy = tm.tmHeight + tm.tmExternalLeading;
        size.cx = m_text.GetLength() * tm.tmAveCharWidth;
        pDC->SelectObject(pOldFont);
    }
    else
    {
        TEXTMETRIC tm;
        pDC->GetTextMetrics(&tm);
        size.cy = tm.tmHeight + tm.tmExternalLeading;
        size.cx = m_text.GetLength() * tm.tmAveCharWidth;
    }
}
```

Voil[aca]! Your OLE server application is ready. Test it by running it at least once; then it can be registered as an OLE server application (See Figure 18.8).

Figure 18.8. An OLE 2.0- (and soon DocObject-) compliant server application.

Modify Your Application to Make It an Operational DocObjects Server Application

This section describes how to merge the BINDSCRB sample with the OLE-created server application. This is a five step process:

1. 1. Insert BINDSCRB sample files into the AppWizard project.
2. 2. Alter a class name to have DocObject- derived classes instead of OLE-derived classes.
3. 3. Add/alter strings in the resource string table.
4. 4. Add a line to have the program registered as a DocObject server application.

5. 5. Set compiling options.

Inserting BINDSCRB Sample Files into the Project

Copy the following files from the BINDSCRB sample (found on the Visual C++ 4 CD-ROM at D:\MSDEV\SAMPLES\MFC\OLE\BINDSCRB) into your project directory:

```
BINDDCMT.CPP

BINDDOC.CPP

BINDIPFW.CPP

BINDITEM.CPP

BINDTARG.CPP

BINDVIEW.CPP

MFCBIND.CPP

OLEOBJCT.CPP

PRINT.CPP

BINDDOC.H

BINDIPFW.H

BINDITEM.H

MFCBIND.H
```

These files implement all necessary interfaces and registration functions for a DocObject server.

Include the .CPP files in your project. Select Files into Project[el] from the Insert menu bar. Select all the preceding .CPP files and validate them. You don't need to include the .H files because they are automatically included.

Altering Classes

The key to integrating the BINDSCRB sample DocObject into your project is altering some class derivation. At least some of your project classes must no longer derive directly from OLE classes. Instead, they must derive from DocObject classes, which derive from OLE classes.

Therefore, you must change the following classes:

```
class CInPlaceFrame: public COleIPFrameWnd
```

```
class CDoServerDoc: public COleServerDoc

class CDoServerSrvItem: public COleServerItem
```

into:

```
class CInPlaceFrame: public CDocObjectIPFrameWnd

class CDoServerDoc: public CDocObjectServerDoc

class CDoServerSrvItem: public CDocObjectServerItem
```

However, you can't just alter these lines; you must also add the correct BINDSCRB header files, change the class names in the IMPLEMENT_DYNCREATE() macros, and so on. All these changes are described in detail in this section.

In the sample provided with this book, all added or altered lines of code for DocObject implementation are presented like this:

```
// DocObject Addition

    // DocObject Modification
```

In the IpFrame.h file, add the following bold lines:

```
#include "BINDIPFW.H"

class CInPlaceFrame: public CDocObjectIPFrameWnd

{

...
```

In the IpFrame.cpp file, modify the class names (see the bold lines):

```
IMPLEMENT_DYNCREATE(CInPlaceFrame, CDocObjectIPFrameWnd)

BEGIN_MESSAGE_MAP(CInPlaceFrame, CDocObjectIPFrameWnd)

    //{{AFX_MSG_MAP(CInPlaceFrame)

    ON_WM_CREATE()

    //}}AFX_MSG_MAP

END_MESSAGE_MAP()
```

In the DoServerDoc.h file, add the following bold lines:

```
#include "BINDDOC.H"

class CDoServerSrvrItem;
```

```
class CDoServerDoc: public CDocObjectServerDoc
{
...
```

In the DoServerDoc.cpp file, modify the class names (see the bold lines):

```
IMPLEMENT_DYNCREATE(CDoServerDoc, CDocObjectServerDoc)
BEGIN_MESSAGE_MAP(CDoServerDoc, CDocObjectServerDoc)
    //{{AFX_MSG_MAP(CDoServerDoc)
        // NOTE—the ClassWizard will add and remove mapping
        // macros here.
        // DO NOT EDIT what you see in these blocks of
        // generated code!
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

In the SrvrItem.h file, add the following bold lines:

```
#include "BINDITEM.H"
class CDoServerSrvrItem: public CDocObjectServerItem
{
```

In the SrvrItem.cpp file, modify the class names (even in the class constructor):

```
IMPLEMENT_DYNAMIC(CDoServerSrvrItem, CDocObjectServerItem)
CDoServerSrvrItem::CDoServerSrvrItem(CDoServerDoc* pContainerDoc)
: CDocObjectServerItem(pContainerDoc, TRUE)
{
    ...
```

Adding and Modifying Resource Text Strings

Some text strings from the project resources must be altered for the program run correctly. Use the resource editor and open

the string table, and then follow these steps:

1. 1. Add the BIND_IDP_FAILED_TO_REGISTER string. In its caption, type Unable to add Binder compatible Registry. This string is used if the function that registers the program fails.
2. 2. Modify the IDR_DOSERVTYPE string caption. Without using a carriage return, type \nDoServ\nDoServ\nDoServer Files (*.dos)\n.DOS\nDoServ.Document.1\nDOS DoServ Document\nDOSE\ndose Files. This string is used to register the program, and is slightly different from the original OLE string.

Registering the Project

Finally, in order to register the server application, add the bold text to DoServer.cpp:

```
#include "stdafx.h"

#include "DoServer.h"

#include "mfcbind.h"

#include "bindipfw.h"

#include "binddoc.h"

...

BOOL CDoServerApp::InitInstance()

{

    ...

    MfcBinderUpdateRegistry(pDocTemplate, OAT_INPLACE_SERVER);

    // m_server.UpdateRegistry(OAT_INPLACE_SERVER);

    ...

}
```

This function automatically registers your project with your OS registry the first time you run the program, thus recognizing it is a DocObject server application.

Setting Compiling Options

Last but not least, you must include the uuid3.lib library in the project. Select Settings[el] from the Build menu bar. Click the Link tab control and enter uuid3.lib in the Object/library modules field (shown in Figure 18.9).

Figure 18.9. Setting the compiling options.

Running and Testing the Project

Compile the project, run it once in order to have it registered, and run the Office Binder. When you select Add[el], you'll see a DoServ document among the choices (see Figure 18.10). Select DoServ to view your DocObject embedded in the Office Binder (see Figure 18.11).

Figure 18.10. DoServer is recognized as a DocObject server application by the Office Binder.

Figure 18.11. A DoServer document embedded within the Office Binder.

How to Create a Basic DocObject Container

This section describes how to create a basic DocObject container from a bare AppWizard application. This two-step process is far easier than creating a DocObject server.

1. 1. Create an OLE container AppWizard application.
- 2. Add the IOleDocumentSite interface.

People who already have an OLE 2.0 container and want to turn it into a DocObject container can skip the first step and go directly to the last one.

Create an AppWizard Application

Create an MFC AppWizard project named DoContainer (you can choose another name; however, I will always refer to DoContainer in the following text). In step 3 of the MFC AppWizard dialog box select the option Container.

The resulting program is an OLE 2.0 container.

Modify It in Order to Have a DocObject Container

Once you have your OLE container, all you have to do is to implement the IOleDocumentSite interface, which contains just one member function. Of course, this is a basic DocObject container. Further interfaces will have to be implemented if you want to fully use DocObject technology.

In the following, the text in bold indicates the code to add (except for the code to add at the end of CntrItem.cpp, where everything is in normal font). The text in normal font represents the code generated by AppWizard.

First, declare the IOleDocumentSite interface within the COleClientItem derived class. In the CntrItem.h file, add

```
#include <docobj.h>

class CDoContainerDoc;

class CDoContainerView;

class CDoContainerCntrItem: public COleClientItem
```

```
{
...
public:
    BEGIN_INTERFACE_PART(DocumentSite, IOleDocumentSite)
        INIT_INTERFACE_PART(CContainerItem, DocumentSite)
        STDMETHOD(ActivateMe)(IOleDocumentView*);
    END_INTERFACE_PART(DocumentSite)
    DECLARE_INTERFACE_MAP()
};
```

Now, at the top of the CntrItem.cpp file, add the MFC implementation of the interface:

```
IMPLEMENT_SERIAL(CDoContainerCntrItem, COleClientItem, 0)
BEGIN_INTERFACE_MAP(CDoContainerCntrItem, COleClientItem)
    INTERFACE_PART(CDoContainerCntrItem, IID_IOleDocumentSite, DocumentSite)
END_INTERFACE_MAP()
CDoContainerCntrItem::CDoContainerCntrItem(CDoContainerDoc* pContainer)
```

Last but not least, add the implementation of the interface member functions at the end of the file.

```
////////////////////////////////////////////////////////////////
// IOleDocumentSite interface
STDMETHODIMP_(ULONG) CDoContainerCntrItem::XDocumentSite::AddRef()
{
    METHOD_PROLOGUE(CDoContainerCntrItem, DocumentSite)
    return pThis->ExternalAddRef();
}
STDMETHODIMP_(ULONG) CDoContainerCntrItem::XDocumentSite::Release()
{
    METHOD_PROLOGUE(CDoContainerCntrItem, DocumentSite)
```

```
    return pThis->ExternalRelease();

}

STDMETHODIMP CDoContainerCntrItem::XDocumentSite::QueryInterface(REFIID iid, LPVOID*
ppvObj)

{

    METHOD_PROLOGUE(CDoContainerCntrItem, DocumentSite)

    return pThis->ExternalQueryInterface(&iid, ppvObj);

}

STDMETHODIMP CDoContainerCntrItem::XDocumentSite::ActivateMe(IOleDocumentView *pView)

{

    METHOD_PROLOGUE(CDoContainerCntrItem, DocumentSite)

    // If there is no view sent, create one

    if (!pView)

      {

        IOleDocument *pDocument;

        if (pThis->m_lpObject->QueryInterface(IID_IOleDocument, (void**)&pDocument)
!= S_OK)

            return E_FAIL;

        if (pDocument->CreateView(&pThis->m_xOleIPSite, NULL, 0, &pView) != S_OK)

            return E_OUTOFMEMORY;

      }

    else

      {

    // Associates the view with the view site

    pView->SetInPlaceSite(&pThis->m_xOleIPSite);

    pView->AddRef();

  }

    // Here, the pView pointer isn't stored.

    // It's up to you to store it if you need it later.
```

```
    // Activate the view's user interface
    pView->UIActivate(TRUE);
    // Retrieves the client coordinates and sets the view port
    RECT  rect;
    GetClientRect(pThis->m_pView->m_hWnd, &rect);
    pView->SetRect(&rect);
    // Displays the view
    pView->Show(TRUE);
    return NOERROR;
}
```

Before compiling, add the uuid3.lib library in the build options. Select Settings[el] from the Build menu bar. Click the Link tab control and enter uuid3.lib in the Object/library modules field (as shown in Figure 18.12).

Compile and execute the program. Then select the Edit menu and choose Insert new object[el]. A dialog box appears and shows all types of objects that can be inserted. In Figure 18.12, the dialog box shows all OLE-compliant servers available (either OLE or DocObject servers). If an OLE-compliant document is selected, the object is inserted within the document already existing. If a DocObject is selected, the DocObject container creates it within its own MDI child window (see Figure 18.13).

Figure 18.12: When inserting an object, the DocObject container shows every OLE document or DocObject available.

Figure 18.13: If a DocObject is inserted, the container inserts it within its own MDI child window.

Summary

OLE Document Objects (DocObjects) represent the latest evolution of OLE, Microsoft's document embedding technology. If this technology is part of ActiveX, whose goal is to «[dg]activate the Internet[dg]», it is because Microsoft sees it as the next Internet standard replacing HTML, Web browsers not displaying HTML pages or Word documents, but DocObjects.

The first part of this chapter has explained what DocObject is and how it works. It then described the full DocObject API, along with how to implement a DocObject server and a DocObject container. Last but not least, it has indicated, step by step, how to create a DocObject server and a DocObject container, either from scratch or from an already existing OLE 2.0 application, thus turning your legacy OLE 2.0 application server or client into a DocObject-compliant server or container application.

Chapter 19

Hyperlink Navigation

Microsoft has decided to open itself to the Internet/intranet world and has oriented its whole strategy around this key idea. Thus, it is already getting its products ready to fully integrate the Internet philosophy:

- Windows 97 will be document- and Internet-oriented, with the Internet Explorer (that is, the Web browser) merged with the Microsoft Explorer (that is, the file manager). Word documents, HTML (HyperText Markup Language) pages, or any type of document should be accessible through this new Explorer, whether they are on the local hard disk, on a corporate network, or on the Internet. Of course, any HTML page can be visualized by the Explorer.
- ActiveX technology was designed for this purpose, too. ActiveX controls are designed to "activate the Internet," that is, to enhance HTML's rather limited functionality.
- OLE Document Object (DocObject for short), part of the ActiveX technology, allows a document to be fully embedded within a container, as if the container were recognizing the DocObject as a native format. Internet Explorer 3.0 (Microsoft's latest state-of-the-art Web browser) already supports this technology, and thus it can display DocObjects (provided the corresponding DocObject server is on the local hard disk) the same way it displays HTML pages.

So far, what does this give you: An Internet-oriented environment where you can display and edit any OLE Document Object stored anywhere on the local hard disk or on the Internet. In Microsoft's vision, DocObject replaces HTML as the standard Web document format. HTML pages are now considered as standard DocObjects (whose server is the Internet Explorer). The next step is to enable every DocObject to provide the functionality that is the base technology of HTML and is responsible for the Web's success: hyperlinking.

A *hyperlink* is a reference to another location (it can be a document, a file, or whatever else) generally represented in an HTML page by colored, underlined text. The user can access this reference by simply clicking on the text. A hyperlink enables you to jump from a document to another through a simple click. Though it was greatly popularized by the World Wide Web, hyperlinking has been used for a long time (for example, in Windows online help).

OLE Hyperlinks enable DocObjects or ActiveX controls to fully support hyperlinking. As a result, an OLE Hyperlinks-compliant document can contain hyperlinks to other existing documents, objects, and applications. This technology also enables a document to use Web browser features.

Definitions

Before going any further, you may want to review the definition of a few key terms, along with an example:

- Hyperlink reference: A reference or pointer to another file or location. A hyperlink is composed of a hyperlink target and a hyperlink location. A *relative reference* is just the path from the hyperlink container to the hyperlink target, whereas an *absolute reference* contains the full path to find the hyperlink target.
- Hyperlink: A hyperlink reference with a friendly name. A hyperlink could be considered as the object that represents what you see: it has a *friendly name* (what is displayed by the user interface), and a reference (the file pointed by the hyperlink).
- Hyperlink container: A document containing a hyperlink reference. A DocObject or an ActiveX control enhanced HTML pages containing hyperlink reference are examples of containers.
- Friendly name: The string that graphically represents the hyperlink on a hyperlink container. The friendly name is the text generally represented in colored underlined text.
- Hyperlink target: A hyperlink container/document/file pointed by a hyperlink reference.
- Location [within the target]: A hyperlink reference can point to a location within a target (like a given bookmark of an HTML page). When no location is defined within the target the hyperlink points to the top of the target document. A hyperlink reference location is coded in HTML as a #*location* (for example, http://www.mysite.com/hello.html#links).
- Hyperlink jump: "Jumping" from the hyperlink container that contains the hyperlink to the hyperlink target.
- Internal jump: The jump is within the same document, but generally at a different location within this document (a jump within the same document and at the same location has no interest).
- Hyperlink navigation: Hyperlink jumping.
- Hyperlink frame: The outer frame that contains the hyperlink container. In this chapter, the hyperlink frame will be referenced as a Web browser for better understanding (especially when the hyperlink frame uses a Web browser's specific features).
- Navigation stack: The stack where the hyperlink frame (that is, a Web browser) writes all the hyperlink references of the previous HTML pages/documents/files displayed. Each time the user has the browser display a new page (either by directly specifying its address or through a hyperlink jump), the browser adds the new page on top of the stack. Whenever the user selects the browser's Go Back command, the navigation stack's current hyperlink reference (that is, the hyperlink reference corresponding to the page being displayed) is moved back one element in the stack (the top element of the stack isn't removed, so that the Go Forward command can be used).
- Hyperlink browse context: Portion of code and data (that is, in implementation terms, objects) of the hyperlink frame that manages the navigation stack.
- Fully integrated hyperlink navigation: Hyperlink navigation using the browse context.
- Browse context's current hyperlink: The hyperlink reference whose hyperlink target is being displayed.

In order to make everything clear, take a look at the concrete example shown in Figure 19.1.

Figure 19.1. A hyperlink frame (Internet Explorer) displaying a hyperlink container (the HTML page).

In Figure 19.1, the hyperlink frame (Internet Explorer 3.0) contains a hyperlink container (the HTML page whose Internet address is http://www.mcp.com/sams/samspub/e-book/30874-6/index.htm, as shown in the *Address* combo box). This page contains several hyperlinks (represented as the underlined blue or red sentences). These hyperlinks are composed of a friendly name (for example, "Topic Quick Reference") and of a hyperlink reference (the relative reference vcuif.htm#l3, as shown in the bottom of the Internet Explorer window). These references point to the hyperlink target (the HTML page whose address is http://www.mcp.com/sams/samspub/e-book/30874-6/vcuif.htm). The location within the target is symbolized by #l3 and points to a hyperlink target's bookmark.

The Internet Explorer also manages a navigation stack, which is the list of all the previously displayed HTML pages (the friendly names of the stack's elements are represented in the *Go* menu bar, as shown in Figure 19.2). This navigation stack is managed by the browse context (some Internet Explorer code and data devoted to this task), thus enabling the user to use buttons such as *Back* or *Forward*.

Figure 19.2. After the jump, the navigation stack has been updated accordingly, as the menu shows (numbers 1 to 4).

The hyperlink navigation performed between Figure 19.1 and 19.2 is a fully integrated one. Indeed, the navigation stack is updated after the jump, as shown in Figure 19.2.

What's the Use of OLE Hyperlinking?

Of course, hyperlinking implementation may not seem to be a big deal. After all, OLE Hyperlink API provides only the hyperlink jump itself; the programmer still has to determine when to launch it (depending on user-created events).

But OLE Hyperlinking first of all allows an easy implementation. Indeed, using the Simple Hyperlink Navigation API, the programmer implements a hyperlink jump with just one function call. Furthermore, using the full Hyperlink Navigation API, the programmer can implement a hyperlink navigation that is fully integrated within an OLE hyperlink-compliant Web browser. That is, when the programmer's OLE hyperlinking-compliant document "jumps" to another document within a hyperlink frame, the jump is notified to the frame's browse context (if any). This means that the navigation stack is updated accordingly, and that the previous document can still be retrieved by using the browser's Go Back command. A hyperlink container can behave like an HTML page. Its URL (Uniform Resource Locator) will be added automatically in the navigation stack.

Furthermore, a hyperlink container can control the hyperlink frame's browse context. It can have access to commands like Go Back or Go Forward, and even to the entire navigation stack (either for visualization purposes or for modifying one).

For example, take a regular HTML page containing a link to a DocObject. In order to go back to the main page, this DocObject can use the browser's Go Back command rather than displaying a hyperlink pointing to the main page. This avoids adding extra copies of the same hyperlink reference in the navigation stack. If such a feature isn't a big deal for an HTML page because browser-widespread scripting languages such as Visual Basic Script or JavaScript provide functions for these types of tasks, OLE Hyperlinking is the only way for a DocObject to access these Web browser's commands (indeed, a DocObject can't call Visual Basic Script or JavaScript code). Furthermore, an ActiveX control may want to directly implement such a feature instead of calling some external script code.

But OLE Hyperlinking provides more functionality than just using the browser's popular commands. Indeed, it provides access to the browser's navigation stack. For example, a DocObject (or an ActiveX control enhanced HTML page) can retrieve the navigation stack, filter its elements that comply to a given set of specifications, and show them to the user. When the user chooses a given hyperlink, the DocObject just sends it to the Web browser (actually, it only sends the hyperlink position within the navigation stack). The browse context will just move the current hyperlink pointer in the stack, without having to reload the referenced hyperlink target.

Hyperlink Navigation API and Simple Hyperlink Navigation API

OLE Hyperlinks come with two APIs: the Hyperlinks Navigation API and the Simple Hyperlink Navigation API. Both have their advantages and their drawbacks and are used for different purposes.

Simple Hyperlink Navigation API

The Simple Hyperlink Navigation API is a reduced set of the full Hyperlink Navigation API and allows basic hyperlinking programming on the hyperlink container side (there is no such thing as simple hyperlinking on the hyperlink frame side). This API is the minimum requirement a DocObject or ActiveX control needs to implement OLE hyperlinking. Even people who are interested in the full Hyperlink Navigation API should first have a look at this API. The advantages of this API are:

- It is easy to implement: because the API is reduced to a few functions, it is not difficult to use, and it is very fast to implement.
- It is not subject to change: this API is not in flux like the full Hyperlink Navigation API, which can be changed by Microsoft at any time. Actually, Microsoft currently encourages developers to use the Simple Navigation API.

The Simple Hyperlink Navigation API consist of six global functions:

```
    HlinkSimpleNavigateToString()

    HlinkSimpleNavigateToMoniker()

    HlinkNavigateString()

    HlinkNavigateMoniker()

    HlinkGoBack()

    HlinkGoForward()
```

It also includes an enumeration:

```
typedef enum tagHLNF {

    HLNF_INTERNALJUMP,

    HLNF_OPENINNEWWINDOW

} HLNF;
```

A Word on the Examples

Because this API is simple, an example is given after each function description. All the examples assume that they are executed within an OLE Document Object (or DocObject). In order to better understand the code, first review some of the argument types that may be required:

- IUnknown*: described as "a pointer to the document or object initiating the hyperlink", this argument may sometimes be optional (that is, it can take a NULL value) and sometimes not. Contrary to what you might think, this argument is not the this pointer cast into a IUnknown* type. It must be a pointer to any interface of the document. Remember that in the ActiveX philosophy, you have a hand on an object if you got a pointer to just one of its interfaces, whatever it is (you can always retrieve the desired interface by calling the IUnknown::QueryInterface() interface member function). In the examples, the value passed is "(IUnknown*)&m_xOleDocument". Indeed, as it is assumed the code is executed within a DocObject, an IOleDocument interface is supposed to be implemented. It means a class is defined to implement this interface, along with a member variable defined as a IOleDocument, which will contain the object interface. If the class is named XOleDocument, the variable is named m_xOleDocument.
- LPCWSTR: This is a string. However, it is not a regular ASCII string, but a Unicode, character strings of 16 bits (LPCWSTR is defined as const unsigned short*). You can't pass a regular C text string. You have first to convert it into Unicode text string by calling ToWideChars(). Note that

there are several other ways to convert ANSI text into Unicode text.

- Moniker*: The creation of a moniker isn't detailed here, as it would mean implementing a whole class. The examples just refer to a moniker variable, defined as a Moniker*.

HlinkSimpleNavigateToString()

This function performs a hyperlink jump. The hyperlink target is determined by a Unicode string.

```
HRESULT HlinkSimpleNavigateToString(LPCWSTR, LPCWSTR, LPCWSTR,
                                    IUnknown*, IBindCtx*,
                                    IBindStatusCallBack*,
                                    DWORD, DWORD);
```

Argument Type Description LPCWSTR Hyperlink target. A NULL value means the navigation is an internal jump. LPCWSTR Location within the hyperlink target (optional, maybe NULL). LPCWSTR Frame within the hyperlink target (optional, maybe NULL). IUnknown* The IUnknown pointer to the document that is initiating the hyperlink. A NULL value means the hyperlink was initiated from a non-OLE-compliant application. IBindCtx* The bind context that will be used for any moniker binding during this navigation. IBindStatusCallback* The bind status callback that will be used for any asynchronous moniker binding during this navigation. A NULL value means that the hyperlink initiator isn't interested in having information such as status or navigation progress. DWORD HLNF enumeration values. DWORD Reserved for a future use. Must be zero.
Returned Value Meaning S_OK The operation succeeded. E_INVALIDARG Some arguments are invalid. Other Other error.

```
    HRESULT hr;
    WCHAR text[21];
    MultiByteToWideChar(CP_ACP, MB_PRECOMPOSED,
                        "D:\\download\\nt40.htm", 21, text, 21);
    hr = HlinkSimpleNavigateToString(text, NULL, NULL,
                                     (IUnknown*)&m_xOleDocument,
                                     NULL, NULL, 0, 0);
```

HlinkSimpleNavigateToMoniker()

This function performs a hyperlink jump. Contrary to HlinkSimpleNavigateToString(), the hyperlink target is not set by a text string but by a moniker.

```
HRESULT HlinkSimpleNavigateToMoniker(IMoniker*, LPCWSTR, LPCWSTR,
                                     IUnknown*, IBindCtx*,
                                     IBindStatusCallBack*,
                                     DWORD, DWORD);
```

Argument Type Description IMoniker* Hyperlink target. A NULL value means the navigation is within a document. LPCWSTR Location within the hyperlink target (optional, maybe NULL). LPCWSTR Frame within the hyperlink target (optional, maybe NULL). IUnknown* The IUnknown pointer to the document that is initiating the hyperlink. A NULL value means the hyperlink was initiated from a non-OLE-compliant application. IBindCtx* The bind context that will be used for any moniker binding during this navigation. IBindStatusCallback* The bind status callback that will be used for any asynchronous moniker binding during this navigation. A NULL value means that the hyperlink initiator isn't interested in having information such as status or navigation progress. DWORD HLNF enumeration values. DWORD Reserved for a future use. Must be zero.
Returned Value Meaning S_OK The operation succeeded. E_INVALIDARG Some arguments are invalid. Other Other error.

```
    HRESULT hr;
    hr = HlinkSimpleNavigateToMoniker(moniker, NULL, NULL,
                                      (IUnknown*)&m_xOleDocument,
                                      NULL, NULL, 0, 0);
```

HlinkNavigateString()

This function is actually a macro that expands itself into a HlinkSimpleNavigateToString() call where most arguments are NULL.

```
HRESULT HlinkNavigateString(IUnknown*, LPCWSTR);
```

Argument Type Description IUnknown* The IUnknown pointer to the document that is initiating the hyperlink. A NULL value means the hyperlink was initiated from a non-OLE-compliant application. LPCWSTR Hyperlink target.
Returned Value Meaning S_OK The operation succeeded. E_INVALIDARG Some arguments are invalid. Other Other error.

```
HRESULT hr;

WCHAR text[21];

MultiByteToWideChar(CP_ACP, MB_PRECOMPOSED,

                    "D:\\download\\nt40.htm", 21, text, 21);

hr = HlinkNavigateString((IUnknown*)&m_xOleDocument, text);
```

HlinkNavigateMoniker()

This function is actually a macro that expands itself into a HlinkSimpleNavigateToMoniker() call where most arguments are NULL.

```
HRESULT HlinkNavigateMoniker(IUnknown*, IMoniker*);
```

Argument Type Description IUnknown* The IUnknown pointer to the document that is initiating the hyperlink. A NULL value means the hyperlink was initiated from a non-OLE-compliant application. IMoniker* Hyperlink target.
Returned Value Meaning S_OK The operation succeeded. E_INVALIDARG Some arguments are invalid. Other Other error.

```
HRESULT hr;

hr = HlinkNavigateString((IUnknown*)&m_xOleDocument, moniker);
```

HlinkGoBack()

This function asks the hyperlink frame (that is, the Web browser) to jump backward in the navigation stack. Here, the only argument that points to the document or object initiating the hyperlink jump cannot be NULL. Furthermore, this function will only work if this same document or object is contained in an OLE hyperlink-compliant container, such as Internet Explorer 3.0.

```
HRESULT HlinkGoBack(IUnknown*);
```

Argument Type Description IUnknown* Pointer to the document or object initiating the hyperlink jump. *Returned Value Meaning* S_OK The operation succeeded. E_INVALIDARG The argument is invalid. Other Other error.

```
    HRESULT hr;

hr = HlinkGoBack((IUnknown*)&m_xOleDocument);
```

HlinkGoForward()

This function asks the hyperlink frame (that is, the Web browser) to jump forward in the navigation stack. Here, the only argument that points to the document or object initiating the hyperlink jump cannot be NULL. Furthermore, this function will only work if this same document or object is contained in an OLE hyperlink-compliant container, such as Internet Explorer 3.0.

```
HRESULT HlinkGoBack(IUnknown*);
```

Argument Type Description IUnknown* Pointer to the document or object initiating the hyperlink jump. *Returned Value Meaning* S_OK The operation succeeded. E_INVALIDARG The argument is invalid. Other Other error.

```
    HRESULT hr;

hr = HlinkGoForward((IUnknown*)&m_xOleDocument);
```

Hyperlink Navigation API

The Hyperlink Navigation API allows much more possibilities than the Simple Hyperlink Navigation API. It is composed of several global functions, enumerations, structures and interfaces. As a result, the HLINK.H file (the hyperlink API include file) from the ActiveX SDK is different from the file provided with Visual C++ 4.1. This means a given program may not compile when the HLINK.H is changed (for example, when the Visual C++ HLINK.H file is replaced by ActiveX SDK's), even if the changes are only minor. You can even expect this file to change during the next Visual C++ releases.

Warning

This API is still in flux and can be modified by Microsoft at any time. Microsoft strongly encourages programmers to use only the simple hyperlink navigation API.

OLE Hyperlink Components

In order to implement the Hyperlink Navigation API, the programmer first needs to understand which are OLE Hyperlink components and how they interact with each other. OLE Hyperlink is composed of two types of programs: the hyperlink frame program and the hyperlink container program. Typically, the hyperlink frame program is a DocObject client and the hyperlink container program is a DocObject server (beware not to mix between a Hyperlink container and a DocObject container). Thus, the frame program can display not only its own native format but also multiple hyperlink container formats.

The Hyperlink Frame Program

This program manages an outer frame that contains and shows hyperlink container. It is called through its IHlinkTarget interface, which must be supported by the class derived from COleClientItem.

This program also contains the hyperlink browse context. This component manages the navigation stack: it notes every hyperlink jump and stores all the containers' references into the stack. It can also allow the user to look at the stack and go to specific hyperlink references within it. The browse context is controlled through its unique IHlinkBrowseContext interface, which must be supported by the class implementing it. This class is typically created for this sole purpose and has its interface pointer given away through the hyperlink frame.

The Hyperlink Container Program

This program manages the hyperlink document being displayed. It can manage either a hyperlink container (that is, a document being displayed) or a hyperlink target (that is, a document pointed at by a hyperlink). For example, the hyperlink container program corresponding to HTML pages is the OLE server part of the Internet Explorer.

This program doesn't always need to implement interfaces. Indeed, it is controlled through the standard OLE and DocObject interfaces. However, if the program wants its documents to be included in the navigation stack by the browse context after a hyperlink jump, it must implement the IHlinkTarget interface.

The hyperlink target can also support the IPersistMoniker interface rather than IPersistFile. Thus, the target can support asynchronous download as a persistence mechanism.

The hyperlink container, as it name implies, contains some hyperlinks and hyperlink sites. A hyperlink site is used by the hyperlink to get the container moniker. Thus, the hyperlink can notice an internal jump by comparing the target moniker to the container moniker and perform the necessary optimizations.

Hyperlinks manage the reference to a document (or more generally any kind of application/document/file). A hyperlink is composed of the hyperlink target (stored as a moniker) and the location within this target (stored as a string). The graphical representation of the hyperlink (such as its friendly name) is up to the container itself. A hyperlink is controlled through its IHlink interface.

How Everything Works Together

This section examines how OLE hyperlink works when a hyperlink navigation occurs. Both simple hyperlink navigation and fully integrated hyperlink navigation will be detailed. Both types of navigation are mainly the same; the last type of navigation just performs some extra work in order to have the browse context integrate the hyperlink references that are navigated to. The diagram below describes the major steps of hyperlink navigation. The numbers indicate the order with which the steps are executed over time and are common to both types of navigation.

As you can see, the hyperlink container calls the hyperlink frame, sending it the desired hyperlink reference interface. The frame then calls the hyperlink reference, which calls the hyperlink target. Then, the target notifies the frame that the navigation successfully performed. This notification is useful for asynchronous navigation, where it is the only way for the frame to know when the navigation is over.

How Simple Hyperlinking Navigation Works

This section describes how the different components interact when a simple hyperlink jump occurs. Of course, only the global function call must be implemented (provided the hyperlink frame already exists). In this example, neither the hyperlink container nor the hyperlink target use (or even know the presence of) the frame's browse context..

1. 1. The Hyperlink container: The hyperlink navigation begins when the hyperlink container calls the global HlinkSimpleNavigateToString() simple hyperlinking navigation API function, which calls the hyperlink frame's IHlinkFrame::Navigate() interface member function. It is up to the hyperlink container to determine whenever it must launch any hyperlink jump, as the container is called by OLE when an event (for example a mouse click on colored, underlined text representing a hyperlink) occurs within the container's view.
2. 2. IHlinkFrame::Navigate(): The frame can perform tasks such as asking confirmation, displaying a progress indicator, managing its hyperlinks history, and so forth. Finally, it calls the hyperlink's IHlink::Navigate() interface member function.
3. 3. IHlink::Navigate(): The hyperlink gives control to the hyperlink target by calling its IHlinkTarget::Navigate() interface member function.
4. 4. IHlinkTarget::Navigate(): The hyperlink target is passed the location string and interprets it (it means the interpretation is at its own discretion). Finally, the target notifies the frame that the hyperlink jump was performed by calling the HlinkOnNavigate() global function, which will call the frame's IHlinkFrame::OnNavigate() member function.
5. 5. IHlinkFrame::OnNavigate(): The frame is notified that the navigation is over and can now update its window.

How Fully Integrated Hyperlink Navigation Works

This kind of navigation allows the integration of navigated hyperlinks within the browse context. All tasks described in the previous section are supposed to be performed here too.

1. 1. The hyperlink container: In order to start the navigation, the container needs three interfaces: the hyperlink to navigate interface, the hyperlink frame's, and its browse context interface.
2. The hyperlink may either be created (through the HlinkCreateFromData(), HlinkCreateFromMoniker() or HlinkCreateFromString() global functions) or loaded from persistent storage (through the OleLoadFromStream() function). The container can then associate it with a hyperlink site by calling its IHlink::SetHlinkSite() member function.
3. The container either retrieves its frame's browse context through the IHlinkFrame::GetBrowseContext() member function or creates it by calling the HlinkCreateBrowseContext() global function. The container registers itself with the browse context by calling the IHlinkBrowseContext::Register() member function. It then calls IHlinkBrowseContext::OnNavigateHlink() to tell the browse context to add the hyperlink to the navigation stack.
4. Finally, it calls the HlinkNavigate() global function, which calls the frame's IHlinkFrame::Navigate() function.
5. 2. IHlinkFrame::Navigate(): Same as with the Simple Hyperlink Navigation API; it calls the hyperlink's IHlink::Navigate() function.
6. 3. IHlink::Navigate(): If passed the HLNF_USERBROWSECONTEXTCLONE value, the hyperlink clones the browse context by calling IHlinkBrowseContext::Clone().

7. For the sake of performance, the hyperlink must check if it is in the presence of an internal jump by comparing the moniker it was passed and the container's moniker returned by the hyperlink site. If the jump is internal, the hyperlink target's interface is retrieved by asking the hyperlink site to return the IHlinkTarget interface of the container (as the container is the target).
8. If the jump is not an internal one, the function must call the target's moniker Imoniker::BindToObject() function to retrieve the target interface, and then call IHlinkTarget::SetBrowseContext(), so that the target registers with the browse context.
9. Eventually, it calls the IHlinkTarget::Navigate() function to perform the jump within the target.
10. 4. IHlinkTarget::SetBrowseContext(): This function is called for a non-internal jump. The target must first revoke any registration with a browse context by calling IHlinkBrowseContext::Revoke() and IHlinkBrowseContext::Release(). Then, the target must register with the browse context passed as an argument by calling IHlinkBrowseContext::AddRef() and IHlinkBrowseContext::Register().
11. IHlinkTarget::Navigate(): This function is called to perform the internal jump part of the hyperlink navigation (even if it is the whole navigation). It is up to the target to interpret the location and to display itself properly. It then calls the IHlinkOnNavigate() global function to notify the hyperlink frame that the navigation was performed. This function calls IHlinkFrame::OnNavigate() and IHlinkBrowseContext::OnNavigateHlink().
12. 5. IHlinkFrame::OnNavigate(): The hyperlink frame now knows that the navigation is over.
13. IHlinkBrowseContext::OnNavigateHlink(): Finally, the browse context is notified that the hyperlink navigation is over, so that it can update the navigation stack accordingly.

The Enumerations/Structures

The interfaces of the Hyperlink Navigation API often use enumerations and/or structures either as input or output arguments. Some of the following enumerations can take several values; some can only take a single value.

The HLNF Enumeration

These values give some information about a hyperlink navigation.
Name Description HLNF_INTERNALJUMP Indicates the navigation is an internal jump. This value allows some internal jump-specific optimization, like avoiding to reload the document. The system will have the hyperlink automatically send this value to IHlink::Navigate() if the moniker is NULL. HLNF_NAVIGATINGBACK Indicates the navigation occurs because the Go Back command was selected. The position in the navigation stack should be moved back one element, without altering the stack. The hyperlink frame and hyperlink container send this value to IHlink::Navigate(). HLNF_NAVIGATINGFORWARD Indicates the navigation occurs because the Go Forward command was selected. The position in the navigation stack should be moved forward one element, without

altering the stack. The hyperlink frame and hyperlink container send this value to IHlink::Navigate(). HLNF_USERBROWSECONTEXTCLONE When IHlink::Navigate() is passed this value, it should clone the browse context passed-in argument by calling IHlinkBrowseContext::Clone(). This new browse context will always be used during the rest of the hyperlink navigation. HLNF_OFFSETWINDOWORG When IHlinkTarget::Navigate() is passed this value, it should alter its window position according to the information contained by the HLBWINFO returned by IHlinkBrowseContext::GetBrowseContextWindowContext(). This value is generally used with the HLNF_USERBROWSECONTEXTCLONE value to implement the Open in new window command. HLNF_OPENINNEWWINDOW Using this value is like using both HLNF_USERBROWSECONTEXTCLONE and HLNF_OFFSETWINDOWORG values. HLNF_CREATENOHISTORY When IHlinkBrowseContext::OnNavigateHlink() is passed this value, it should not add the hyperlink to the navigation stack. HLNF_NAVIGATINGTOSTACKITEM Same as HLNF_CREATENOHISTORY. Furthermore, IHlinkBrowseContext::OnNavigateHlink() should set the passed hyperlink as the browse context's current hyperlink. This value is used when the user decides to jump to a hyperlink from the navigation stack: though the navigation occurs, the navigation stack should not be altered, only the position on the stack.

The HLINKWHICHMK Enumeration

A single value from this enumeration is used by the IHlinkSite::GetMoniker() interface member function to know whether it must return a moniker for the container or a base moniker specific to the site.
Name Description HLINKWHICHMK_CONTAINER Return the moniker for the hyperlink container corresponding to a particular hyperlink site. HLINKWHICHMK_BASE Return the base moniker corresponding to a particular hyperlink site.

The HLINKGETREF Enumeration

A single value from this enumeration is used by the IHlink::GetMonikerReference() and IHlink::GetStringReference() interface functions to know if they must return the absolute or relative reference for the hyperlink target.
Name Description HLINKGETREF_DEFAULT Return the default reference for hyperlink target. HLINKGETREF_ABSOLUTE Return the absolute reference for hyperlink target. HLINKGETREF_RELATIVE Return the relative reference for hyperlink target.

The HLFNAMEF Enumeration

A single value from this enumeration is used by IHlink::GetFriendName() to know which type of friendly name to return.

Name Description HLFNAMEF_TRYCACHE Returns the friendly name that is cached on the Hlink interface object. HLFNAMEF_TRYFULLTARGET Returns the full display name of the hyperlink target. HLFNAMEF_TRYPRETTYTARGET Returns a beautiful version of the full display name of the hyperlink target. HLFNAMEF_TRYWIN95SHORTCUT Returns the version of the full display name of the hyperlink target without any path or extension. HLFNAMEF_DEFAULT Returns the cached friendly name.

The HLINKMISC Enumeration

A single value of this enumeration is returned by IHlink::GetMiscStatus() to tell whether a hyperlink is relative or absolute.
Name Description HLINKMISC_ABSOLUTE The hyperlink contains an absolute reference to the hyperlink target. HLINKMISC_RELATIVE The hyperlink contains a relative reference to the hyperlink target.

The HLITEM Structure

This structure is used by the IEnumHLITEM interface, which belongs to enumerators returned by IHlinkBrowseContext::EnumNavigationStack().
Name Type Description uHLID ULONG The hyperlink ID. szFriendlyName LPWSTR The friendly name of the hyperlink.

The HLBWIF Enumeration

Values from this enumeration are passed in an HLBWINFO structure, which is associated with each browse context. The HLIBWINFO structure is accessed by calling IHlinkBrowseContext::GetBrowseWindowContext() and IHlinkBrowseContext::SetBrowseWindowContext().
Name Description HLBWIF_HASFRAMEWNDINFO The browse context has available frame-level window positioning information. HLBWIF_HASDOCWNDINFO The browse context has available document-level window positioning information. HLBWIF_FRAMEWNDMAXIMIZED The browse context's frame-level windows should appear maximized. Always used with HLBWIF_HASFRAMEWNDINFO. HLBWIF_DOCWNDMAXIMIZED The browse context's document-level windows should appear maximized. Always used with HLBWIF_HASDOCWNDINFO.

The HLBWINFO Structure

This structure contains information about the locations and sizes of frame-level and document-level windows within a browse context. This structure is returned from the browse context by calling IHlinkBrowseContext::GetBrowseWindowContext() and sending to the browse context by calling IHlinkBrowseContext::SetBrowseWindowContext(). This structure is used by a hyperlink target during IHlinkTarget::Navigate() so that it can reposition its user interface.
Name Type Description cbSize ULONG Size of the structure in bytes. grfHLBWIF DWORD HLBWIF enumeration values. rcFramePos RECTL If grfHLBWIF has the FLBWIF_HASFRAMEWNDINFO value, this variable contains the rectangle in screen coordinates of current frame-level windows within the browse context. If grfHLBWIF has the FLBWIF_FRAMEWNDMAXIMIZED value, this means the frame-level windows are currently maximized, and rcFramePos represents the non-maximized size of the document-level windows. rcDocPos RECTL If grfHLBWIF has the FLBWIF_HASDOCWNDINFO value, this contains the rectangle in screen coordinates of current document-level windows within the browse context. If grfHLBWIF has the FLBWIF_DOCWNDMAXIMIZED value, this means the document-level windows are currently maximized, and rcFramePos represents the non-maximized size of the document-level windows.

The HLID Constants

Some specific hyperlinks are identified by an HLID constant rather by a IHlink interface pointer. Thus, the hyperlink frame can perform specific optimization.
Name Description HLID_PREVIOUS Indicates the hyperlink before the current one in the navigation stack. If such a hyperlink does not exist (either the current hyperlink is the first one in the navigation stack or the stack is empty), methods like IHlinkBrowseContext::GetHlink() return NULL and E_FAIL. HLID_NEXT Indicates the hyperlink past the current one in the navigation stack. If such a hyperlink does not exist (either the current hyperlink is the last one in the navigation stack or the stack is empty), methods like IHlinkBrowseContext::GetHlink() return NULL and E_FAIL. HLID_CURRENT Indicates the current hyperlink in the navigation stack. This value can be used when the hyperlink has to be physically navigated again. For example, a page can have to be reloaded (to implement a Reload command), a sound sample may have to be replayed, and so on. HLID_STACKBOTTOM Indicates the first hyperlink of the navigation stack. If the stack is empty, methods like IHlinkBrowseContext::GetHlink() return NULL and E_FAIL. HLID_STACKTOP Indicates the last hyperlink of the navigation stack. If the stack is empty, methods like IHlinkBrowseContext::GetHlink() return NULL and E_FAIL.

The HLQF Enumeration

A single value of this enumeration is used by IHlinkBrowseContext::QueryHlink() to know which kind of hyperlink test it must perform.
Name Description HLQF_ISVALID Indicates to test the validity of a particular hyperlink.

HLQF_ISCURRENT Indicates to test if a particular hyperlink is the browse context's current hyperlink.

The HLSR Enumeration

A single value from this enumeration is used by the HlinkGetSpecialReference() or HlinkSetSpecialReference() global functions to know which value to set or get.
Name Description HLSR_HOME Hyperlink reference to the user's home page. HLSR_SEARCHPAGE Hyperlink reference to the user's search page. HLSR_HISTORYFOLDER Hyperlink reference to the user's history folder page.

The CF_HYPERLINK Clipboard Format

This format consists of a serialized hyperlink. When used with IDataObject, this format is passed under TYMED_ISTREAM or TYMED_HGLOBAL mediums. The OleSaveToStreamEx() function stores a hyperlink into an IStream, provided the hyperlink supports the OLE IPersistStream interface.

The Global Functions

The full Hyperlink Navigation API provides a set of global functions. This allows the creation of a hyperlink from different types of data, and the initiation of a hyperlink navigation.

HlinkCreateBrowseContext()

This function creates an empty browse context instance, whose interface is returned through the last argument.

```
HRESULT HlinkCreateBrowseContext(IUnknown*, REFIID, void**);
```

Argument Type Description IUnknown* IUnknown interface of the object controlling the new browse context. A NULL value (the most common) means the browse context isn't aggregated. REFIID Which type of browse context's interface to return, typically IID_IHlinkBrowseContext. void** Where to send the browse context REFIID interface.
Returned Value Meaning S_OK The browse context was successfully created. E_OUTOFMEMORY Not enough memory for creating the browse context. E_INVALIDARG At least one argument is invalid.

HlinkQueryCreateFromData()

This function determines if a hyperlink can be created from the IDataObject interface passed-in argument, that is, if:

- The IDataObject offers CF_HYPERLINK on either TYMED_ISTREAM or TYMED_HGLOBAL.
- The IDataObject offers Win95 shortcut data (this format isn't yet determined).

Here, the result is returned by the function itself, and not through any argument.

```
HRESULT HlinkQueryCreateFromData(IDataObject*);
```

Argument Type Description IDataObject* The source data interface.
Returned Value Meaning S_OK Yes, a hyperlink can be created from the given IDataObject E_OUTOFMEMORY No, a hyperlink can't be created because the IDataObject doesn't fulfill the requirements. E_INVALIDARG The argument is invalid.

HlinkCreateFromData()

This function creates a hyperlink from the IDataObject passed as the first argument. The created hyperlink is returned through the last argument.

This function can be called after a cut-and-paste operation from the clipboard, in which case the IDataObject interface is retrieved by calling OleGetClipboard(). It can be also called after a drag-and-drop operation, in which case the IDataObject interface is retrieved by calling IDropTarget::Drop(), where the IDropTarget interface is the one registered to the target drag-and-drop window.

```
HRESULT HlinkCreateFromData(IDataObject*, IHlinkSite*, DWORD,

                            IUnknown*, REFIID, void**);
```

Argument Type Description IDataObject* Source data from which the hyperlink must be created. IHlinkSite* The hyperlink site associated to the new hyperlink. DWORD Additional hyperlink site data. IUnknown* IUnknown interface of the object controlling the new hyperlink. A NULL value (the most common) means the hyperlink object isn't aggregated. REFIID Which hyperlink interface must the function return, generally IID_IHlink. void** Where to send the hyperlink desired interface.
Returned Value Meaning S_OK The hyperlink was successfully created. E_NOINTERFACE The

hyperlink does not support the desired interface. E_INVALIDARG At least one argument is invalid.

HlinkCreateFromMoniker()

This function creates a hyperlink from a moniker whose IMoniker interface is passed as the first argument. The created hyperlink is returned through the last argument. This function is faster than HlinkCreateFromString() if the moniker is already defined.

It is generally called when the program wants to create a hyperlink from an already existing hyperlink. It can retrieve the already existing hyperlink's moniker's interface and target location through Hlink::GetMonikerReference() and its friendly name through GetFriendlyName().

```
HRESULT HlinkCreateFromMoniker(IMoniker*, LPCWSTR, LPCWSTR,

                               IHlinkSite*, DWORD, IUnknown*,

                               REFIID, void**);
```

Argument Type Description IMoniker* The interface of the moniker from which the hyperlink must be created. LPCWSTR Location of the new hyperlink within the hyperlink target. May not be NULL. LPCWSTR The new hyperlink friendly name. IHlinkSite* The hyperlink site to be associated to the hyperlink. DWORD Additional hyperlink site data. IUnknown* IUnknown interface of the object controlling the new hyperlink. A NULL value (the most common) means the hyperlink object isn't aggregated. REFIID The hyperlink's interface to be returned, generally IID_IHlink. void** Where to return the desired hyperlink interface.
Returned Value Meaning S_OK The hyperlink was successfully created. E_INVALIDARG At least one argument is invalid.

HlinkCreateFromString()

This function creates a hyperlink from strings representing the hyperlink target, the location within this target, and the friendly name. The created hyperlink is returned through the last argument. This function is slower than HlinkCreateFromMoniker() as it must parse the text strings and create a moniker. The HlinkCreateFromMoniker() function should be called if the hyperlink target's moniker already exists.

```
HRESULT HlinkCreateFromString(LPCWSTR, LPCWSTR, LPCWSTR,

                              IHlinkSite*, DWORD, IUnknown*,
```

```
REFIID, void**);
```

Argument Type Description LPCWSTR String describing the hyperlink target. LPCWSTR Location of the new hyperlink within the hyperlink target. May not be NULL. LPCWSTR The new hyperlink friendly name. IHlinkSite* The hyperlink site to be associated to the hyperlink. DWORD Additional hyperlink site data. IUnknown* IUnknown interface of the object controlling the new hyperlink. A NULL value (the most common) means the hyperlink object isn't aggregated. REFIID The hyperlink's interface to be returned, generally IID_IHlink.
Returned Value Meaning S_OK The hyperlink was successfully created. E_INVALIDARG At least one argument is invalid.

HlinkGetSpecialReference()

This function retrieves the current user's global home, search, or history page as a string. The result is returned through the second argument, and it can be turned into a hyperlink by calling the HlinkCreateFromString() global function.

```
HRESULT HlinkGetSpecialReference(DWORD, LPCWSTR*);
```

Argument Type Description DWORD A HLSR enumeration value. Determines if the function must return the home page, search page, or history page. LPWSTR* Where to send the required page as a string.
Returned Value Meaning S_OK The function retrieved the page successfully. E_INVALIDARG At least one argument is invalid.

HlinkSetSpecialReference()

This function sets the current user's global home, search, or history page as a string.

```
HRESULT HlinkSetSpecialReference(DWORD, LPCWSTR);
```

Argument Type Description DWORD A HLSR enumeration value. Determines if the function must set the home page, search page, or history page. LPCWSTR Where to send the required page as a string.
Returned Value Meaning S_OK The operation succeeded. E_INVALIDARG At least one argument is invalid.

HlinkNavigateToStringReference()

This function performs a hyperlink navigation by regrouping common function calls: It creates a hyperlink from a string by calling HlinkCreateFromString(), performs hyperlink navigation by calling HlinkNavigate(), and then releases the hyperlink by calling Hlink::Release().

```
HRESULT HlinkNavigateToStringReference(LPCWSTR, LPCWSTR,
                        IHlinkSite*, DWORD, IHlinkFrame*, DWORD,
                        IBindCtx*, IBindStatusCallback*
                        IHlinkBrowseContext*);
```

Argument Type Description LPCWSTR String describing the hyperlink target. LPCWSTR Location within the hyperlink target. IHlinkSite* The hyperlink site to be associated to the hyperlink. DWORD Additional site data. IHlinkFrame* Hyperlink frame of the hyperlink container. A NULL value means the container doesn't have any frame. DWORD HLNF enumeration values. IBindCtx* The bind context to use for every moniker that will be created during the navigation. May not be NULL. IBindStatusCallback* The bind status callback to use for asynchronous moniker binding. A NULL value means the caller doesn't need to know information such as cancellation, progress state, and so on. IHlinkBrowseContext* The browse context to use for this navigation.
Returned Value Meaning S_OK The hyperlink jump was successfully performed. E_INVALIDARG At least one argument is invalid.

HlinkNavigate()

This function calls IHlinkFrame::Navigate() if it is passed a hyperlink frame as argument, and calls IHlink::Navigate() if it is only passed a hyperlink as argument.

```
HRESULT HlinkNavigate(IHlink*, IHlinkFrame*, DWORD, IBindCtx*,
                        IBindStatusCallback*, IHlinkBrowseContext*);
```

Argument Type Description IHlink* The hyperlink to navigate to. IHlinkFrame* The hyperlink frame of the hyperlink container. A NULL value means the hyperlink container doesn't have any frame. DWORD HLNF enumeration values. IBindCtx* The bind context to use for every moniker that will be created during the navigation. May not be NULL. IBindStatusCallback* The bind status callback to use for asynchronous moniker binding. A NULL value means the caller doesn't need to know information such as cancellation, progress state, and so forth. IHlinkBrowseContext* The browse context to use for this navigation.
Returned Value Meaning S_OK The hyperlink navigation succeeded. E_INVALIDARG At least one

argument is invalid.

HlinkOnNavigate()

This function encapsulates functions during IHlinkTarget::Navigate() execution. It calls IHlinkBrowserContext::OnNavigateHlink() and IHlinkFrame::OnNavigate() if the hyperlink target has a hyperlink frame.

```
HRESULT HlinkOnNavigate(IHlinkFrame*, IHlinkBrowseContext*, DWORD,

                        IMoniker*, LPCWSTR, LPCWSTR);
```

Argument Type Description IHlinkFrame* Hyperlink frame of the hyperlink container. A NULL value means the container doesn't have any frame. IHlinkBrowseContext* The browse context to use for this navigation. DWORD HLNF enumeration values. IMoniker* The hyperlink target's moniker. May not be NULL. LPCWSTR Location within the hyperlink target. LPCWSTR The friendly name of the hyperlink. *Returned Value Meaning* S_OK The hyperlink navigation succeeded. E_INVALIDARG At least one argument is invalid.

The IHlinkSite Interface

This interface is used by a hyperlink site and allows the site's associated hyperlink to retrieve information about the container.

```
BEGIN_INTERFACE_PART(HlinkSite, IHlinkSite)

    INIT_INTERFACE_PART(CMyDocObject, HlinkSite)

    STDMETHOD(GetMoniker)(DWORD, DWORD, DWORD, Moniker**);

    STDMETHOD(GetInterface)(DWORD, DWORD, REFIID, void**);

    STDMETHOD(OnNavigateComplete)(DWORD, HRESULT, LPCWSTR);

END_INTERFACE_PART(HlinkSite)
```

IHlinkSite::GetMoniker()

This member function retrieves the moniker of the hyperlink site's container, which is returned through the last argument.

```
STDMETHOD(GetMoniker)(DWORD, DWORD, DWORD, Moniker**);
```

Argument Type Description DWORD Hyperlink associated with the site. This value was given by IHlink::SetHlinkSite(). DWORD An OLEGETMONIKER enumeration value. An OLEGETMONIKER_ONLYIFTHERE value means the function must not create a moniker, even if it doesn't already exist. An OLEGETMONIKER_FORCEASSIGN value means the function should create a moniker if it doesn't already exist. DWORD An OLEWHICHMK enumeration value. An OLEWHICHMK_CONTAINER value means the hyperlink site must return the container's moniker. IMoniker** Where to send the container's moniker interface pointer.
Returned Value Meaning S_OK The moniker was successfully returned. E_INVALIDARG At least one argument is invalid.

IHlinkSite::GetInterface()

This member function retrieves a hyperlink site's container interface, which is returned through the last argument.

If a hyperlink, while comparing the target's moniker to the container's moniker (returned by IHlinkSite::GetMoniker()), finds that the navigation is an internal jump, it retrieves the hyperlink target's IHlinkTarget interface by calling this function.

This function differs slightly from IUnknown::QueryInterface(). First, it may return an interface based on the hyperlink passed-in argument. Second, the returned interface may not belong to the hyperlink site.

```
STDMETHOD(GetInterface)(DWORD, DWORD, REFIID, void**);
```

Argument Type Description DWORD Hyperlink associated with the site. This value was given by IHlink::SetHlinkSite(). DWORD Reserved for future use. Must be zero. REFIID The desired interface's IID. void** Where to send the container's desired interface pointer.
Returned Value Meaning S_OK The interface was successfully returned. E_NOINTERFACE The desired interface isn't supported by the container. E_INVALIDARG At least one argument is invalid.

IHlinkSite::OnNavigationComplete()

This member function is called whenever a hyperlink jump has been performed. This function can be useful when the jump is executed asynchronously, because it's the only way for the site to know when the navigation has completed.

```
STDMETHOD(OnNavigateComplete)(DWORD, HRESULT, LPCWSTR);
```

Argument Type Description DWORD Hyperlink associated with the site. This value was given by Ihlink::SetHlinkSite(). HRESULT Result of the hyperlink navigation. Must be S_OK, E_ABORT or E_FAIL. LPCWSTR A text string describing, in case of a problem, the failure that occurred. *Returned Value Meaning* S_OK Success. E_INVALIDARG At least one argument is invalid.

The IHlink Interface

This interface is used by a hyperlink. It allows hyperlink-specific information management (that is, either retrieving or setting) such as its moniker, its hyperlink site, its friendly name, and so forth.

```
BEGIN_INTERFACE_PART(Hlink, IHlink)

    INIT_INTERFACE_PART(CMyDocObject, Hlink)

    STDMETHOD(GetHlinkSite)(IHlinkSite**, DWORD*);

    STDMETHOD(SetHlinkSite)(IHlinkSite*, DWORD);

    STDMETHOD(GetMonikerReference)(DWORD, IMoniker**, LPWSTR*);

    STDMETHOD(GetStringReference)(DWORD, LPWSTR*, LPWSTR*);

    STDMETHOD(GetFriendlyName)(DWORD, LPCWSTR*);

    STDMETHOD(SetFriendlyName)(LPCWSTR);

    STDMETHOD(GetTargetFrameName)(LPCWSTR*);

    STDMETHOD(SetTargetFrameName)(LPCWSTR);

    STDMETHOD(GetAdditionalParams)(LPCWSTR*);

    STDMETHOD(SetAdditionalParams)(LPCWSTR);
```

```
    STDMETHOD(Navigate)(DWORD, IBindCtx*, IBindStatusCallback*,
                        IHlinkBrowseContext*);
    STDMETHOD(GetMiscStatus)(DWORD*);
END_INTERFACE_PART(Hlink)
```

IHlink::GetHlinkSite()

This member function retrieves the hyperlink's associated site, along with its data, which are returned through the two arguments.

```
STDMETHOD(GetHlinkSite)(IHlinkSite**, DWORD*);
```

Argument Type Description IHlinkSite** Where to send the hyperlink associated site's interface. A NULL value is *not* accepted. DWORD* Where to send further site data. A NULL value is *not* accepted. *Returned Value Meaning* S_OK The hyperlink site was successfully returned. E_INVALIDARG At least one argument is invalid.

IHlink::SetHlinkSite()

This member function associates the hyperlink with a hyperlink site whose interface is passed in the first argument. A hyperlink must be associated with a hyperlink site in order to work properly when IHlink::Navigate() is called.

```
STDMETHOD(SetHlinkSite)(IHlinkSite*, DWORD);
```

Argument Type Description IHlinkSite* Hyperlink site's interface. DWORD Further site's data. *Returned Value Meaning* S_OK The hyperlink site was successfully associated. E_INVALIDARG At least one argument is invalid.

IHlink::GetMonikerReference()

This member function retrieves the container's moniker and the location within the target container, which are returned through the last two arguments.

Implementation note: the moniker passed in the second argument can bind to the target by calling IMoniker::BindToObject(). Once the hyperlink jump succeeded, this function can perform an internal hyperlink jump to go to the location, by calling Hlink::Navigate().

```
STDMETHOD(GetMonikerReference)(DWORD, IMoniker**, LPWSTR*);
```

Argument Type Description DWORD An HLINKGETREF enumeration value. IMoniker** Where to send the container's moniker interface pointer. LPWSTR* Location within the hyperlink target.
Returned Value Meaning S_OK The moniker was successfully retrieved. E_INVALIDARG At least one argument is invalid.

IHlink::GetStringReference()

This member function retrieves the container's address and location as strings, which are returned through the last two arguments.

```
STDMETHOD(GetStringReference)(DWORD, LPWSTR*, LPWSTR*);
```

Argument Type Description DWORD A HLINKGETREF enumeration value. LPWSTR* Where to send the string. LPWSTR* Where to send the location.
Returned Value Meaning S_OK The string reference was successfully retrieved. E_INVALIDARG At least one argument is invalid.

IHlink::GetFriendlyName()

This member function retrieves the friendly name of the hyperlink reference, which is returned through the last argument. This function is called by the hyperlink container when updating its user interface.

Note that the friendly name returned from the hyperlink may differ from the one returned from the target, as the hyperlink may cache it.

```
STDMETHOD(GetFriendlyName)(DWORD, LPCWSTR*);
```

Argument Type Description DWORD An HLFNAMEF enumeration value. Indicates which friendly

name must be returned. LPWSTR* Where to send the friendly name.
Returned Value Meaning S_OK The friendly name was successfully returned. E_INVALIDARG At least one argument is invalid.

IHlink::SetFriendlyName()

This member function sets the friendly name of a hyperlink reference.

```
STDMETHOD(SetFriendlyName)(LPCWSTR);
```

Argument Type Description LPCWSTR Where to send the friendly name.
Returned Value Meaning S_OK The friendly name was successfully set. E_INVALIDARG Invalid argument.

IHlink::GetTargetFrameName()

This member function retrieves the name of the target frame (as in HTML framesets). The name is returned through the single argument and is useless if the container doesn't support frame-sets.

```
STDMETHOD(GetTargetFrameName)(LPCWSTR*);
```

Argument Type Description LPWSTR* Where to send the target frame name.
Returned Value Meaning S_OK The target frame name was successfully returned. E_INVALIDARG Invalid argument.

IHlink::SetTargetFrameName()

This member function sets the target frame name. The name is useless if the container doesn't support frame-sets.

```
STDMETHOD(SetTargetFrameName)(LPCWSTR);
```

Argument Type Description LPCWSTR Target frame name.

Returned Value Meaning S_OK The target frame name was successfully set. E_INVALIDARG Invalid argument.

IHlink::GetAdditionalParams()

This member function retrieves additional parameters or properties of the hyperlink, which are returned through the single argument. The parameter string's format is designed as follows:

```
<ID1="value1"><ID2="value2">...<Idn="valuen">
```

Most of the parameters saved in this string are specific to the hyperlink frame.

```
STDMETHOD(GetAdditionalParams)(LPCWSTR*);
```

Argument Type Description LPWSTR* Where to send the parameter string.
Returned Value Meaning S_OK The parameter string was successfully retrieved. E_INVALIDARG Invalid argument.

IHlink::SetAdditionalParams()

This member function sets additional parameters or properties of the hyperlink. The string format is described in IHlink::SetAdditionalParams().

```
STDMETHOD(SetAdditionalParams)(LPCWSTR);
```

Argument Type Description LPCWSTR Parameter string.
Returned Value Meaning S_OK The parameters were successfully passed. E_INVALIDARG At least one argument is invalid.

IHlink::Navigate()

This member function perform a hyperlink jump.

```
STDMETHOD(Navigate)(DWORD, IBindCtx*, IBindStatusCallback*,

                    IHlinkBrowseContext*);
```

Argument Type Description DWORD HLNF enumeration values. IBindCtx* The bind context to use for every moniker that will be created during the navigation. May not be NULL. IBindStatusCallback* The bind status callback to use for asynchronous moniker binding. A NULL value means the caller doesn't need to know information such as cancellation, progress state, and so on. IHlinkBrowseContext* The browse context that will be used during this hyperlink navigation.
Returned Value Meaning S_OK The navigation succeeded. HLINK_S_NAVIGATEDTOLEAFNODE TBD. E_INVALIDARG At least one argument is invalid.

IHlink::GetMiscStatus()

Asks if the hyperlink is an absolute or relative link. The answer is returned through the single argument.

```
STDMETHOD(GetMiscStatus)(DWORD*);
```

Argument Type Description DWORD* Where to send an HLINKMISC enumeration value.
Returned Value Meaning S_OK Success. E_INVALIDARG Invalid argument.

The IHlinkTarget Interface

This interface is used to access a hyperlink container, typically when it is viewed as a hyperlink target (hence the interface name).

```
BEGIN_INTERFACE_PART(HlinkTarget, IHlinkTarget)

    INIT_INTERFACE_PART(CMyDocObject, HlinkTarget)

    STDMETHOD(GetBrowseContext)(IHlinkBrowseContext**);

    STDMETHOD(SetBrowseContext)(IHlinkBrowseContext*);

    STDMETHOD(Navigate)(DWORD, IBindCtx*, IBindStatusCallback*,

                        IHlink*);
```

```
    STDMETHOD(GetMoniker)(LPCWSTR, DWORD, IMoniker**);

    STDMETHOD(GetFriendlyName)(LPCWSTR, LPWSTR*);

END_INTERFACE_PART(HlinkTarget)
```

IHlinkTarget::GetBrowseContext()

This member function retrieves the browse context of the target. This function returns the desired interface through the argument and calls HlinkBrowseContext::AddRef() in order to tell the browse context the target is holding a reference to it.

This function doesn't need to be implemented for simple hyperlinking.

```
STDMETHOD(GetBrowseContext)(IHlinkBrowseContext**);
```

Argument Type Description IHlinkBrowseContext** Where to send the browse context interface. *Returned Value Meaning* S_OK The browse context was successfully returned. E_NOTIMPL Browse context retrieval not supported. E_INVALIDARG At least one argument is invalid.

IHlinkTarget::SetBrowseContext()

This member function sets the hyperlink target's browse context. This function will typically release its previous browse context (if any) by calling IHlinkBrowseContext::Release() and hold a reference to the browse context passed-in argument by calling IHlinkBrowseContext::AddRef().

This function doesn't need to be implemented for simple hyperlinking.

```
STDMETHOD(SetBrowseContext)(IHlinkBrowseContext*);
```

Argument Type Description IHlinkBrowseContext* The new browse context interface pointer. *Returned Value Meaning* S_OK The browse context is successfully set. E_NOTIMPL The hyperlink target does not understand browse context. E_INVALIDARG At least one argument is invalid.

IHlinkTarget::Navigate()

This function navigates to the location and shows it if not visible (that is, display it on screen). Note that if the target supports browse context, it calls IHlinkBrowseContext::OnNavigateHlink() to notify a hyperlink jump.

This function doesn't need to be implemented for simple hyperlinking.

```
STDMETHOD(Navigate)(DWORD, IBindCtx*, IBindStatusCallback*,
                    IHlink*);
```

Argument Type Description DWORD HLNF enumeration values. LPCWSTR Location within the hyperlink target.
Returned Value Meaning S_OK The navigation succeeded. E_INVALIDARG At least one argument is invalid.

IHlinkTarget::GetMoniker()

This member function returns a moniker pointing to the hyperlink target, for the given hyperlink location. The moniker's interface is returned through the last argument.

```
STDMETHOD(GetMoniker)(LPCWSTR, DWORD, IMoniker**);
```

Argument Type Description LPCWSTR Location within the hyperlink target. DWORD An OLEGETMONIKER enumeration. Must be either OLEGETMONIKER_ONLYIFTHERE (if the moniker does not exist, return E_FAIL) or OLEGETMONIKER_FORCEASSIGN (if the moniker does not exist, create it). IMoniker** Where to send the resulting moniker.
Returned Value Meaning S_OK The moniker was successfully returned. E_FAIL The moniker does not exist and the OLEGETMONIKER_ONLYIFTHERE flag was set. E_INVALIDARG At least one argument is invalid.

IHlinkTarget::GetFriendlyName()

This member function retrieves the friendly name, which is returned through the last argument.

```
STDMETHOD(GetFriendlyName)(LPCWSTR, LPWSTR*);
```

Argument Type Description LPCWSTR The location within the target. LPWSTR* Where to return the friendly name. This string must be allocated by the function by calling CoTaskMemFree() and freed by the function's caller through CoTaskMemFree().
Returned Value Meaning S_OK The friendly name was successfully retrieved. E_OUTOFMEMORY Not enough memory to create the friendly name string. E_INVALIDARG At least one argument is invalid.

The IHlinkFrame Interface

This interface is used by the hyperlink frame. It is mostly used for browse context management (either retrieving or setting the browse context).

```
BEGIN_INTERFACE_PART(HlinkTarget, IHlinkTarget)

    INIT_INTERFACE_PART(CMyDocObject, HlinkTarget)

    STDMETHOD(GetBrowseContext)(IHlinkBrowseContext**);

    STDMETHOD(SetBrowseContext)(IHlinkBrowseContext*);

    STDMETHOD(Navigate)(DWORD, IBindCtx*, IBindStatusCallback*,

                        IHlink*);

    STDMETHOD(OnNavigate)(DWORD);

END_INTERFACE_PART(HlinkTarget)
```

IHlinkFrame::GetBrowseContext()

This member function retrieves the hyperlink frame browse context, which is returned through the function's single argument.

This function doesn't need to be implemented for simple hyperlinking.

```
STDMETHOD(GetBrowseContext)(IHlinkBrowseContext**);
```

Argument Type Description IHlinkBrowseContext** Where to send the browse context interface pointer.

Returned Value Meaning S_OK The browse context was successfully retrieved. E_NOTIMPL Browse context retrieval not supported. E_INVALIDARG At least one argument is invalid.

IHlinkFrame::SetBrowseContext()

This member function sets the hyperlink frame browse context.

This function doesn't need to be implemented for simple hyperlinking.

```
STDMETHOD(SetBrowseContext)(IHlinkBrowseContext*);
```

Argument Type Description IHlinkBrowseContext* The browse context interface.
Returned Value Meaning S_OK The given browse context was successfully set. E_NOTIMPL Browse context setting not supported. E_INVALIDARG At least one argument is invalid.

IHlinkFrame::Navigate()

This member function performs a hyperlink navigation. It is called by the IHlink::Navigate() interface member function when it is given a non-NULL IHlinkFrame pointer. The goal of this function is to allow the hyperlink frame to perform some action during the hyperlink jump.

```
STDMETHOD(Navigate)(DWORD, IBindCtx*, IBindStatusCallback*,

                    IHlink*);
```

Argument Type Description DWORD HLNF enumeration values. IBindCtx* The bind context to use for every moniker that will be created during the hyperlink jump. IBindStatusCallback* The bind status callback to use for every asynchronous moniker that will be created during the hyperlink jump. IHlink* The hyperlink to navigate to.
Returned Value Meaning S_OK The hyperlink jump succeeded. E_INVALIDARG At least one argument is invalid. Other From IHlink::Navigate().

IHlinkFrame::OnNavigate()

This member function is called to notify the hyperlink frame that the navigation succeeded. This allows a hyperlink frame to update its window. This function is called by the hyperlink target during

IHlinkTarget::Navigate() using the global function HlinkOnNavigate().

```
STDMETHOD(OnNavigate)(DWORD);
```

Argument Type Description DWORD HLNF enumeration values.
Returned Value Meaning S_OK Success. E_INVALIDARG Invalid argument.

The IHlinkBrowseContext Interface

This interface is returned by the hyperlink frame and is used for browse context management.

```
BEGIN_INTERFACE_PART(HlinkBrowseContext, IHlinkBrowseContext)
   INIT_INTERFACE_PART(CMyDocObject, HlinkBrowseContext)
   STDMETHOD(Register)(DWORD, IUnknown*, IMoniker*, DWORD*);
   STDMETHOD(GetObject)(IMoniker*, IUnknown**);
   STDMETHOD(Revoke)(DWORD);
   STDMETHOD(GetBrowseWindowInfo)(HLBWINFO*);
   STDMETHOD(SetBrowseWindowInfo)(HLBWINFO*);
   STDMETHOD(EnumNavigationStack)(IEnumHLITEM**);
   STDMETHOD(QueryHlink)(ULONG);
   STDMETHOD(GetHlink)(ULONG, Iklink**);
   STDMETHOD(SetCurrentHlink)(ULONG);
   STDMETHOD(OnNavigateHlink)(DWORD, IMoniker*, LPCWSTR, LPCWSTR);
   STDMETHOD(Clone)(IUnknown*, REFIID, IUnknown**);
   STDMETHOD(Close)(DWORD);
END_INTERFACE_PART(HlinkBrowseContext)
```

IHlinkBrowseContext::Register()

This member function registers an object with the browse context. The browse context maintains a table of moniker-object bindings. This table can be accessed for navigation to already registered hyperlink targets. When a hyperlink jump occurs, the browse context looks at this table and checks if the hyperlink target is already registered and is running. If so, it doesn't have to reload the same document object. This function returns a registration ID (through the last argument), which can be used to revoke the registration.

```
STDMETHOD(Register)(DWORD, IUnknown*, IMoniker*, DWORD*);
```

Argument Type Description DWORD Reserved for future use. Must be zero. IUnknown* The object to register. IMoniker* The moniker that points to the object being registered. DWORD* Where to send the output value, that is, the registration ID.
Returned Value Meaning S_OK The object was successfully registered. MK_S_MONIKERALREADYREGISTERED The object was successfully registered. However, another object has already been registered with the same moniker. E_OUTOFMEMORY Not enough memory to register the object. E_INVALIDARG At least one argument is invalid.

IHlinkBrowseContext::GetObject()

This member function retrieves the object corresponding to the moniker passed in the first argument. The resulting object is returned through the last argument.

```
STDMETHOD(GetObject)(IMoniker*, IUnknown**);
```

Argument Type Description IMoniker* Identifies the object to retrieve. IUnknown** Where to send the object's IUnknown interface pointer.
Returned Value Meaning S_OK The object was successfully retrieved. S_FALSE There was no object corresponding to the moniker being passed. E_INVALIDARG At least one argument is invalid.

IHlinkBrowseContext::Revoke()

This member function cancels the object whose registration ID is the one passed in argument. This ID is

given by IHlinkBrowseContext::Register().

```
STDMETHOD(Revoke)(DWORD);
```

Argument Type Description DWORD Registration ID of the object to be revoked.
Returned Value Meaning S_OK The object was successfully removed from registration.
E_INVALIDARG At least one argument is invalid.

IHlinkBrowseContext::GetBrowseWindowInfo()

This member function retrieves the HLBWINFO structure currently associated with the browse context, which is returned through the only function argument. This function is generally called by a hyperlink target during IHlinkTarget::Navigate() in order to draw its user interface.

```
STDMETHOD(GetBrowseWindowInfo)(HLBWINFO*);
```

Argument Type Description HLBWINFO* Where to return the HLBWINFO structure containing window information.
Returned Value Meaning S_OK The HLDWINFO structure was successfully returned.
E_INVALIDARG Invalid argument.

IHlinkBrowseContext::SetBrowseWindowInfo()

This member function is the opposite of IHlinkBrowseContext::GetBrowseWindowInfo(). It sets the browse context window information from a HLBWINFO structure passed-in argument. This function is generally used by the hyperlink target and the hyperlink container when the window is resized.

```
STDMETHOD(SetBrowseWindowInfo)(HLBWINFO*);
```

Argument Type Description HLBWINFO* The HLBWINFO structure containing window information.
Returned Value Meaning S_OK The window settings were updated. E_INVALIDARG Invalid argument.

IHlinkBrowseContext::EnumNavigationStack()

This member function creates an HLITEM enumerator that will browse a sequence of HLITEM structures. The sequence structures will contain information about the navigation stack's hyperlinks and friendly names associated. This function returns through the argument the enumerator's interface, that is IEnumHLITEM.

```
STDMETHOD(EnumNavigationStack)(IEnumHLITEM**);
```

Argument Type Description IEnumHLITEM** Where to send the enumerator's interface.
Returned Value Meaning S_OK The enumerator was successfully created. E_INVALIDARG Invalid argument.

IHlinkBrowseContext::QueryHlink()

This member function tests the validity of a hyperlink. It is generally called by the hyperlink frame to determine the validity of commands such as Go Forward or Go Back by passing HLID_NEXT and HLID_PREVIOUS. The HLQF enumeration must be either HLQF_ISVALID or HLQF_ISCURRENT. A HLQF_ISVALID value means the function must check if the hyperlink (whose ID is passed in the second argument) is valid within the browse context. A HLQF_ISCURRENT value means the function must check if the hyperlink is the current browse context hyperlink.

```
STDMETHOD(QueryHlink)(ULONG);
```

Argument Type Description DWORD A HLQF enumeration value. ULONG Identifies the hyperlink to check. Can be an HLID constant.
Returned Value Meaning S_OK The test succeeded. E_FALSE The test failed. E_INVALIDARG The HLQF value is invalid.

IHlinkBrowseContext::GetHlink()

This member function retrieves from the browse context the hyperlink whose ID is passed as the first argument. The resulting hyperlink IHlink interface is returned through the second argument.

```
STDMETHOD(GetHlink)(ULONG, Iklink**);
```

Argument Type Description ULONG Identifies the hyperlink to retrieve. Can be an HLID constant.

IHlink** Where to send the interface of the hyperlink to retrieve.
Returned Value Meaning S_OK The hyperlink was successfully retrieved. E_FAIL The desired hyperlink does not exist. E_INVALIDARG Invalid IHlink** argument.

IHlinkBrowseContext::SetCurrentHlink()

This member function sets as current hyperlink in the browse context the hyperlink whose ID is passed in argument.

```
STDMETHOD(SetCurrentHlink)(ULONG);
```

Argument Type Description ULONG Identifies the hyperlink to set as current browse context hyperlink. Can be an HLID constant.
Returned Value Meaning S_OK The hyperlink was successfully set as current hyperlink. E_FAIL The desired hyperlink does not exist. E_INVALIDARG Invalid argument.

IHlinkBrowseContext::OnNavigateHlink()

This member function is called by the hyperlink target during IHlinkTarget::Navigate() to notify the browse context that a hyperlink jump has been performed.

```
STDMETHOD(OnNavigateHlink)(DWORD, IMoniker*, LPCWSTR, LPCWSTR);
```

Argument Type Description DWORD HLNF enumeration values. IMoniker* The moniker of the hyperlink target. LPCWSTR The location within the hyperlink target. May not be NULL. LPCWSTR The friendly name of the location within the hyperlink target. May not be NULL.
Returned Value Meaning S_OK Success. E_INVALIDARG At least one argument is invalid.

IHlinkBrowseContext::Clone()

This member function clones the browse context. The clone is returned through the last argument.

```
STDMETHOD(Clone)(IUnknown*, REFIID, IUnknown**);
```

Argument Type Description IUnknown* IUnknown interface of the object controlling the new browse context. A NULL value (the most common) means the browse context isn't aggregated. REFIID Interface ID of the created browse context. Generally IID_IHlinkBrowseContext. void** Where to return the new browse context desired interface.
Returned Value Meaning S_OK The new browse context was successfully created. E_INVALIDARG At least one argument is invalid.

IHlinkBrowseContext::Close()

This member function closes the browse context. It releases all the hyperlink targets that are still registered through IHlinkBrowseContext::Register().

```
STDMETHOD(Close)(DWORD);
```

Argument Type Description DWORD Reserved for a future use. Must be zero.
Returned Value Meaning S_OK The browse context has been successfully closed. E_INVALIDARG Invalid argument.

The IEnumHLITEM Interface

This interface is a standard OLE enumeration interface. When the IHlinkBrowseContext::EnumNavigationStack() function is called, it creates a HLITEM structures enumerator and returns its IEnumHLITEM interface. The created enumerator is controlled through this interface and can browse a HLITEM sequence.

```
BEGIN_INTERFACE_PART(EnumHLITEM, IEnumHLITEM)

    INIT_INTERFACE_PART(CMyDocObject, EnumHLITEM)

    STDMETHOD(Next)(ULONG, HLITEM*, ULONG*);

    STDMETHOD(Skip)(ULONG);

    STDMETHOD(Reset)(void);

    STDMETHOD(Clone)(IEnumHLITEM**);
```

```
END_INTERFACE_PART(EnumHLITEM)
```

IEnumHLITEM::Next()

This member function retrieves a given number (passed as the first argument) of HLITEM structures. These structures will be copied in an HLITEM array (passed as the second argument).

```
STDMETHOD(Next)(ULONG, HLITEM*, ULONG*);
```

Argument Type Description ULONG Number of HLITEM structures to retrieve. HLITEM* Array where the HLITEM structures will be copied. ULONG* Where to send the actual number of HLITEM copied, in case there are fewer elements left than requested.
Returned Value Meaning S_OK The structures were successfully returned. S_FALSE Some structures were successfully returned. However, there were fewer elements left than requested. E_INVALIDARG At least one argument is invalid.

IEnumHLITEM::Skip()

This member function skips a given number of elements.

```
STDMETHOD(Skip)(ULONG);
```

Argument Type Description ULONG The number of elements to skip.
Returned Value Meaning S_OK The given number of elements were skipped. S_FALSE There were fewer elements left in the sequence than requested. The enumerator is now at the end of the sequence.

IEnumHLITEM::Reset()

This member function sets the enumerator at the beginning of the sequence.

```
STDMETHOD(Reset)(void);
```

Returned Value Meaning S_OK Success.

IEnumHLITEM::Clone()

This member function clones the enumerator. The clone will have the same state, that is the same position of the sequence.

```
STDMETHOD(Clone)(IEnumHLITEM**);
```

Argument Type Description IEnumHLITEM** Where to send the created enumerator's interface pointer. *Returned Value Meaning* S_OK The enumerator was successfully cloned. E_OUTOFMEMORY Not enough memory to create a new enumerator. E_INVALIDARG Invalid argument.

Summary

OLE Hyperlinking Navigation brings to ActiveX that which made the Web successful: Hyperlinks, that is, how to jump from one document to another by a simple mouse click. This technology allows either a DocObject or an ActiveX control to simply implement hyperlinks and to interact with the Web Browser. Hyperlinking Navigation comes with two sets of APIs. The first set is a simple API and enables quick hyperlinking implementation. The second API, which is far larger, implements many more features, such as a greater interaction with the Web browser than with the simple API. However, this API is tougher to implement and, most of all, still subject to change.

This chapter first explains exactly what OLE Hyperlinking Navigation is, how it works, and what you can do with it. It then describes what the differences are between the two APIs. Finally, it explains in detail each API.

Appendix A

Internet Explorer 3.0

by Weiying Chen

Microsoft Internet Explorer 3.0 provides a number of revolutionary features to the Web users. These features include ActiveX content, personalization capability, Internet mail, news, chatting, and Internet security support.

ActiveX content features include HTML enhancement, ActiveX controls used inside IE 3.0, ActiveX scripting use including VBScript and JavaScript, ActiveX document support so that the Document object such as Microsoft Word can be seamlessly embedded in the IE 3.0.

This appendix will focus on the ActiveX content features and along with a discussion on the other features such as Internet security, personalization, and communication. Each of the ActiveX content features will be presented in detail and reflected in the example used next.

ActiveX Content Feature Showcase

Let's take a look at the Web page shown in Figure A.1.

Figure A.1. ActiveX Content Feature showcase.

Note

To view this page, please create a directory called xa under wwwroot and then copy all the *.htm, *.gif, *.dll and *.doc files contained in the CD to that directory. And make sure redir.dll is under scripts directory. redir.dll does not include this CD because it is not created by me, but it can be found in a lot of places. Enter http://yourmachine/xa/a1.htm in the address edit box in the IE 3.0.

There are three frames in this page. IE 3.0 supports frame which includes bordered, borderless, targeted window, and floating frame. Frame allows you to divide the page into independent windows. Each independent window is actually a browse. Look at the source code for this page in Listing A.1.

Listing A.1. a1.htm source code.

```
<HTML>

<HEAD>

<TITLE>

ActiveX Programming Unleashed - Appendix A

</TITLE>

</HEAD>

<BODY BGCOLOR=#FFFFFF TOPMARGIN=0 LEFTMARGIN=0 VLINK=#000080>

<FRAMESET rows = "25%, *" FRAMEBORDER=0>

<FRAME name="top" src ="/xa/a2.htm" scrolling="NO">

<FRAMESET cols="55%,*" FRAMEBORDER = 0>

    <FRAME name = "leftbottom" marginwidth=5 marginheight=5  SCROLLING ="yes"

[icc]src="/xa/a3.htm">

    <FRAME name = "rightbottom" marginwidth=10 marginheight=10 SCROLLING="yes"

[icc]src="/xa/a4.htm">

</FRAMESET>

</FRAMESET>

</BODY>

</HTML>
```

In Listing A.1, Tag FRAMESET defines layout of frames within a page. The attribute rows divides the page into rows.

In this case, there are two rows. cols divides the page into columns. 25% means that the first row will occupy 25% of the window. FRAME tag defines the independent windows. The window style can be specified by using attributes such as MARGINWIDTH, MARGINHEIGHT, SCROLLING, FRAMEBORDER, and so on. NAME attribute can be used to specify the frame name. There are three frames in the frameset, named top, leftbottom, rightbottom.

ActiveX Controls and ActiveX Scripting

The frame named top consists of five elements, its source address is a2.htm. This frame demonstrates ActiveX content, including the use of VBScript and JavaScript, object model exposed by IE 3.0, ActiveX control used in VBScript, aligning text in adjacent cells by baseline and marquee.

The first element is the .GIF about Internet Explorer. The second element is the greeting (good morning, good afternoon, and good evening) depending on when the page is viewed, which is written in JavaScript. The third is to display your machine name, which is implemented by VBScript and an ActiveX control named lst44.dll. lst44.dll is created in Chapter 4, "Creating OLE Automation Servers." The fourth is the .GIF dotted line. The last one is a scrolling marquee using marquee tag.

Let's take a look at Listing A.2 for the source code (a2.htm) of the frame named top.

Listing A.2. a2.htm source code.

```
<HTML>

<HEAD>

<OBJECT   ID="MachineName"

     CLASSID="clsid:C566CC25-182E-11D0-A6AD-00AA00602553"

    CODEBASE="http://weiying1/xa/lst44.dll"

    ALIGN="baseline" BORDER="0" WIDTH="0" HEIGHT="0"

    TYPE="application/x-oleobject">

</OBJECT>

</HEAD>

<BODY BGCOLOR=#FFFFFF TOPMARGIN=0 LEFTMARGIN=0 VLINK=#000080>

<TABLE CELLPADDING=3 BORDER=0 CELLSPACING=0>

<TR VALIGN=TOP >

<TD>

<IMG SRC="/xa/ie.gif"  WIDTH=360 HEIGHT=60>
```

```
</TD>
<FONT SIZE=4>
<TD VALIGN=MIDDLE>
<SCRIPT LANGUAGE="JavaScript">
<!--
    var date = new Date()
    var hour = date.getHours()
    if (hour < 12)
        document.write("Good morning!")
    else
        if (hour < 17)
            document.write("Good afternoon!")
        else
            document.write("Good evening!")
-->
</SCRIPT>
<SCRIPT LANGUAGE="VBSCRIPT">
    document.write "<br>Your machine name is:"
    document.write MachineName.getMachineName
</SCRIPT>
</FONT>
</TD>
</TR>
</TABLE>
<IMG SRC="dot.gif" WIDTH=600 HEIGHT=1>
<FONT SIZE=2 COLOR="green"><MARQUEE BEHAVIOR=SLIDE DIRECTION=LEFT BORDER="0">
```

```
Microsoft Internet Explorer 3.0 with ActiveX technologyt</MARQUEE></FONT>

<IMG SRC="dot.gif" WIDTH=600 HEIGHT=1>

</BODY>

</HTML>
```

In Listing A.2, tag OBJECT inserts an OLE control; its ID is MachineName. CLASSID identifies the object implementation. CODEBASE identifies the codebase for the object. TYPE indicates the internet media type.

Observant readers will notice the following code. Note that the following code is abridged from Listing A.2 to emphasize the essential parts:

```
<TD VALIGN=MIDDLE>

<SCRIPT LANGUAGE="JavaScript">
```

Attribute VALIGN for tag TD will align text in adjacent cells by baseline. This is a features provided by IE 3.0 for enhanced tables. The other features for enhanced tables are rows and columns grouping, selectable borders, and background-images placement in individual table cells.

Tag SCRIPT specifies the inclusion of a script. The attribute LANGUAGE indicates in which ActiveX scripting language the enclosed script is written. There are two popular scripting languages: VBScript and JavaScript.

VBScript and JavaScript both support a set of built-in functions. For instance,

```
var date = new Date()
```

date is an object provided by JavaScript.

In addition to the built-in functions, the ActiveX scripting languages can also access the object model exposed by IE 3.0. For instance, in the following code:

```
<SCRIPT LANGUAGE="VBSCRIPT">

    document.write "<br>Your machine name is:"

</SCRIPT>
```

document is the object exposed by the IE 3.0. write is a method exposed by the document object. Besides document object, there are other objects such as window, form, navigator, history, and so forth exposed by the IE 3.0. All these objects expose a set of methods, properties, or events. IE 3.0 also provides a set of intrinsic controls, which is used as the input in the form tag. The intrinsic controls include checkbox, hidden, image, password, radio, reset, submit, and text.

VBScript and JavaScript can also use the functionality exposed by the ActiveX controls. For instance:

```
document.write MachineName.getMachineName
```

MachineName is an OLE automation server. getMachineName is the method exposed by this automation server. MachineName.getMachineName will return the name of the machine where the browser is running.

HTML Enhancement and ActiveX Document

The frame named leftbottom demonstrates the use of VBScript, Meta tag, object model exposed by IE 3.0, ActiveX document, floating frames. Leftbottom frame consists of six elements: three .GIF and three floating frames. Take a look at Listing A.3 for the a3.htm source code.

Listing A.3. a3.htm source code.

```
<BODY BGCOLOR=#FFFFFF TOPMARGIN=0 LEFTMARGIN=0 VLINK=#000080>

<TABLE BORDER=0  ALIGN=left CELLSPACING=5>

<TR>

<TD ALIGN=left VALIGN=TOP>

<IMG SRC="/xa/active.gif">

</TD>

</TR>

<TR>

<TD>

<iFRAME SRC="/xa/active.htm" NAME="float1" WIDTH=400 HEIGHT=110 FRAMEBORDER=0>

<FRAME SRC="/xa/active.htm" NAME="float1" WIDTH=400 HEIGHT=110 FRAMEBORDER=0>

</iFRAME>

</TD>

</TR>

<TR>

<TD ALIGN=left VALIGN=TOP>

<IMG  SRC="/xa/secure.gif" ALT="Bullet">

</TD>

</TR>
```

```
<TR>
<TD>
<iFRAME SRC="/xa/secure.doc" NAME="float2" WIDTH=400  HEIGHT=110 FRAMEBORDER=0>
<FRAME SRC="/xa/secure.doc" NAME="float2" WIDTH=400  HEIGHT=110 FRAMEBORDER=0>
</iFRAME>
</TD>
</TR>
<TR>
<TD ALIGN=left VALIGN=TOP>
<IMG  SRC="/xa/more.gif" ALT="Bullet">
</TD>
</TR>
<TR>
<TD>
<iFRAME SRC="http://www.microsoft.com/ie/ie3/" NAME="float3" WIDTH=400 HEIGHT=
[icc]110 FRAMEBORDER=0>
<FRAME SRC="http://www.microsoft.com/ie/ie3/" NAME="float3" WIDTH=400  HEIGHT=
[icc]110 FRAMEBORDER=0>
</iFRAME>
</TD>
</TR>
</TABLE>
```

In Listing A.3, tag IFRAME indicates a "floating" window in a web page. The floating frame can be inserted into any place where an image can be put in a conventional browser. There are three floating frames in Listing A.3. They are called float1, float2, and float3.The SRC attribute specifies the source address for the floating frame. The source for float1 is active.htm as indicated in the following code:

```
<FRAME SRC="/xa/active.htm" NAME="float1" WIDTH=400 HEIGHT=110 FRAMEBORDER=0>
```

The source for float2 is secure.doc as indicates as follows:

```
 <FRAME SRC="/xa/secure.doc" NAME="float2" WIDTH=400  HEIGHT=110 FRAMEBORDER=0>
```

The source for float3 is http://www.microsoft.com/ie/ie3/ as indicated as follows:

```
<FRAME SRC="http://www.microsoft.com/ie/ie3/" NAME="float3" WIDTH=400  HEIGHT=110
FRAMEBORDER=0>
```

Different source will take different effect. Float1 will display different content every 20 seconds. Float2 is a Word document that can be seamlessly placed in the browser. Besides Word documents, Microsoft Excel and Visio documents can be placed in the browser seamlessly, too. The reason is that IE 3.0 is a document object container, whereas Word, Excel, and Visio are document objects. Any document object can be seamlessly placed inside a document object container. Float3 displays the web page at http://www.microsoft.com/ie/ie3/. You can click link in float3. The reason is that a frame is actually a browser.

The source for float1 is active.htm. Listing A.4 will explain how float1 works. active.htm demonstrates the following ActiveX content features: style sheet, VBScript, object model exposed by IE 3.0, and Meta tag.

Listing A.4. active.htm source code.

```
<html>

<head>

<META HTTP-EQUIV="Refresh" CONTENT="20; URL=active.htm">

<STYLE>

    BODY {font: 12pt/13pt Arial; color: white}

</STYLE>

<title>VBScript Sample</title>

</head>

<body bgcolor=green  topmargin=0 leftmargin=0>

<script language="vbscript">

<!--

On Error Resume Next

DIM Feature(10)
```

```
lowerbound=0
i=0
Feature= "Authoring Features, new  HTML Enhancement including Style Sheet,
[icc]Table Enhancement, Frames etc"
i=i+1
Feature="ActiveX controls  including ""legacy"" control, OLE automation
[icc]server, and Java Applet used in IE 3.0"
i=i+1
Feature="ActiveX Scripting including VBScript and JavaScript"
i=i+1
Feature="Internet Server Framework including ISAPI and Server side
[icc]Scripting"
i=i+1
Feature ="ActiveX Document enables document object embedded in IE 3.0
[icc]seamlessly"
i=i+1
Feature="Active VRML, ActiveMovie"
i=i+1
Feature="Compatible with Netscape Plug in technology"
Randomize()
pick = Int((i + 1) * Rnd )
QA = Feature(pick)
document.write "<TD BGCOLOR=green>"
document.write QA
document.write "</font>"
-->
```

```
</script>

</BODY>

</HTML>
```

In Listing A.4, tag META provides information about an HTML document using a Name/Value pair to the browsers. There are four attributes used by META tag. They are HTTP-EQUIV, CONTENT, NAME, and URL. The CONTENT attribute specifies the value for the name/value pair. NAME specifies the name of the name/value pair. HTTP-EQUIV binds the content element to an HTTP header field. URL indicates the document's URL. The following code will tell the browser to refresh the content from active.htm every 20 seconds:

```
<META HTTP-EQUIV="Refresh" CONTENT="20; URL=active.htm">
```

Observant readers notice the following code:

```
<STYLE>

    BODY {font: 12pt/13pt Arial; color: white}

</STYLE>
```

Tag STYLE specifies the rendering information. Information enclosed between the beginning STYLE tag and ending STYLE tag overrides client defaults. Attribute font groups font-family, font-size, line-height, font-weight, font-style together. The font-weight and font-style must be specified before other font attributes. This code

```
{font: 12pt/13pt Arial; color: white}
```

can be expanded into

```
P {font-family: Arial;

    font-size: 12pt;

    line-height: 13pt};
```

For more information on style sheets, please refer to Appendix F, "HTML Enhancement by Internet Explorer 3.0."

Forms and ISAPI

The rightbottom frame demonstrates the use of ISAPI DLL, forms, and tables. When the Download button is clicked with option IE 3.0 for Windows 95, the corresponding URL will be navigated to. Figure A.2 demonstrates that frames can be used as a browser.

Figure A.2. Redirect to a URL.

Listing A.5 demonstrates the ActiveX content feature including internet server framework, particularly, ISAPI DLL, and forms.

Listing A.5.a4.htm source code.

```
<BODY BGCOLOR=BLACK  TOPMARGIN=0 LEFTMARGIN=0 VLINK=#000080>

<TABLE>

<TR>

<FORM NAME="IEFamily" METHOD=GET ACTION="/scripts/redir.dll">

<FONT FACE="ARIAL" SIZE=2>

<IMG SRC="/xa/hot.gif" WIDTH=22 HEIGHT=14>

Internet Explore Family<BR>

<SELECT NAME="Target">

<OPTION VALUE="http://www.microsoft.com/msdownload/ie/11.htm">IE 3.0 for

[icc]Windows 95

<OPTION VALUE="http://www.microsoft.com/msdownload/ie/10.htm">IE 3.0 for

[icc]Windows NT 4.0

<OPTION VALUE="http://www.microsoft.com/msdownload/ie/09.htm">IE 3.0 Patch

[icc]for Windows 95 & NT 4.0

<OPTION VALUE="http://206.204.65.109/cgi-bin/ieitar.pl">IE 3.0 128 Bit Version

<OPTION VALUE="http://www.microsoft.com/msdownload/ie/07.htm">IE 2.0 for

[icc]Windows 95

<OPTION VALUE="http://www.microsoft.com/msdownload/ie/0600.htm">IE 2.0 for

[icc]Windows NT(i386)

<OPTION VALUE="http://www.microsoft.com/msdownload/ie/0500.htm">IE 2.0 for

[icc]Windows NT(PPC)

<OPTION VALUE="http://www.microsoft.com/msdownload/ie/0400.htm">IE 2.0 for

[icc]Windows NT(MIPS)
```

```
<OPTION VALUE="http://www.microsoft.com/msdownload/ie/0300.htm">IE 2.0 for
[icc]Windows NT(Dec Alpha)
<OPTION VALUE="http://www.microsoft.com/msdownload/ie/02.htm">IE 2.1 for
[icc]Windows 3.1
<OPTION VALUE="http://www.microsoft.com/msdownload/ie/01.htm">IE 2.1 for
[icc]Macintosh
<OPTION VALUE="http://www.microsoft.com/msdownload/ie/00.htm">IE 2.01 for
[icc]Windows 3.1
<OPTION VALUE="http://www.microsoft.com/ie/download/ieadd.htm">Additional
[icc]Features & Add-ons
</SELECT><P>
</FONT>
<INPUT VALUE="Download" TYPE=submit>
</FORM>
</TR>
<TR>
<BR>
<BR>
<BR>
<FORM NAME="AddsOn" METHOD=GET ACTION="/scripts/redir.dll">
<FONT FACE="ARIAL" SIZE=2>
<IMG SRC="/xa/hot.gif" WIDTH=22 HEIGHT=14>
Additional Features and Add-ons<BR>
<SELECT NAME="Target">
<OPTION VALUE="http://www.microsoft.com/msdownload/ieadd/10.htm">NetMeeting for
[icc]Windows 95
```

```
<OPTION VALUE="http://www.microsoft.com/msdownload/ieadd/09.htm">Internet Mail
[icc]and News for Windows 95
<OPTION VALUE="http://www.microsoft.com/msdownload/ieadd/08.htm">Internet Mail
[icc]and News for Windows NT 4.0
<OPTION VALUE="http://www.microsoft.com/msdownload/ieadd/0100.htm">Comic Chat
<OPTION VALUE="http://www.microsoft.com/msdownload/ieadd/03.htm">Citrix
[icc]WinFrame Web Client
<OPTION VALUE="http://www.microsoft.com/msdownload/ieadd/07.htm">ActiveMovie
[icc]1.0
<OPTION VALUE="http://www.microsoft.com/msdownload/ieadd/0000.htm">DirectX 2
[icc]for Windows 95
<OPTION VALUE="http://www.microsoft.com/msdownload/ieadd/0400.htm">VRML 1.0
[icc]ActiveX Control
<OPTION VALUE="http://www.microsoft.com/msdownload/ieadd/0600.htm">Macromedia
[icc]Shockwave ActiveX Control
<OPTION VALUE="http://www.microsoft.com/msdownload/ieadd/0500.htm">HTML Layout
[icc]Control
<OPTION VALUE="http://www.microsoft.com/msdownload/ieadd/02.htm">IE 3.0
[icc]International Extensions
</SELECT><P>
</FONT>
< INPUT VALUE="Download" TYPE=submit>
</FORM>
</TR>
</TABLE>
```

In Listing A.5, tag FORM denotes a form. There are three attributes used by a FORM tag. They are ACTION, METHOD, and TARGET. The value associated with ACTION is an address(called action URL) to carry out the action

of the form. METHOD indicates how the form data should be sent to the server. It can be GET or POST. GET will append the argument to the action URL. And the server can retrieve the value of the argument via QUERY_STRING environment variable. POST sends the form data to the server. TARGET means to load the results into the targeted window.

Observant readers notice that some tags are very frequently used, such as HTML, HEAD, BODY, FORM, TABLE, TD, TR, SCRIPT, OBJECT, MARQUEE, FRAME, STYLE, and FONT. For detailed information on HTML tags, please refer to http://www.microsoft.com/workshop/author/newhtml/.

IE Family and Add-Ons

From Listing A.5, the Internet Explorer family is indicated in the form IEFamily. Internet Explorer has the following versions:

- IE 3.0 on Windows 95, Windows NT 4.0, 128 bitVersion.
- IE 2.0 on Windows 95, Windows NT (i386, PPC, MIPS, DEc Alpha), Windows 3.1, Macintosh, and IE 2.01 for Windows 3.1.

The minimum installation only includes the browser. A typical installation includes the Internet Mail and News plus the following additional features and add-ons list. Full installation includes Internet Mail and News, NetMeeting for Windows 95, ActiveMovie, and the HTML Layout Control.

The form AddsOn enumerates the additional features and Add-ons in case more software is needed beyond the full installation. The add-ons are

- NetMeeting for Windows 95 which supports running Web phone, white board, and chat across the internet.
- Internet Mail and News for Windows 95 and NT 4.0.
- Comic Chat to proivde internet chats with comic-strip style.
- Citrix WinFrame Web Client is an ActiveX control. It supports running Windows application over the Web.
- ActiveMovie 1.0 control.
- DirectX 2 for Windows 95.
- Macromedia Shockwave ActiveX control.
- HTML Layout Control, which supports pop-up windows, interact by dragging and dropping objects, and animated effect.
- IE 3.0 International Extensions.

Personalization, Internet Security, and Communication Feature

Along with the ActiveX content feature support by IE 3.0, personalization, internet security, and internet communication, such as mail, news, and chat are supported by the browser itself or by installing the add-on software.

Personalization capability allows Web users to change and size toolbar and linkbar so that parents can take control of their children by selecting a different level to access the Web, and can use the keyboard to surf the Web.

Internet Security includes the following items. Authenticode provides the ability to verify the source before download happens. Client authentication via digital personal certificate enables you to securely identify yourself to others on the

Internet. Secure communication protocol such as Secure Socket Layer(SSL) 2.0, 3.0, and Private Communications Technology(PCT) enables you to do internet banking and on-line stock trading.

Summary

IE 3.0 with the ActiveX technology provides an innovative browser to the Web user. Creating dazzling Web pages becomes reality, using the browser becomes easier, and surfing the Web is much safer.

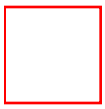
Appendix B

Microsoft Internet Explorer Logo Program

Microsoft released the Internet Explorer Logo Program, which appeared last year, and whose principle is based on the Netscape logo program. It enables any Webmaster to insert a Microsoft logo like this in his or her Web site:

Best experienced with

Click here to start



provided the Web site complies to several rules. The goal of this program is to promote Internet Explorer as *the* Web browser, and at the same time promote ActiveX technology.

Microsoft also provides a complete set of free tools to help people achieve IE Logo Program-compliant HTML pages.

What Does the Logo Mean?

Two types of Internet Explorer logo can be inserted in a Web page: a static logo and an animated one. The static logo indicates that the Web site is best viewed with Microsoft Internet Explorer because it uses some of its extra features (a bit like the Netscape logo, which indicates that the site uses Netscape-defined HTML features such as frames). The animated logo also indicates that the concerned Web site takes advantage of ActiveX technology and includes at least one ActiveX-compatible control.

General Guidelines

Microsoft issued a series of guidelines concerning both logos, which everyone using the logo should follow:

- Intent: The logo must be used only in order to promote Internet Explorer. It must not be used in any way that may harm Microsoft, or criticize its products, nor use it as a form of Microsoft sponsorship or endorsement. It must not be used to promote products that could be deemed pornographic, violent, or unlawful.
- Logo presentation: Microsoft strongly encourages designers that the logo should be represented within text like "Best experienced with ... Click here to start" (see the example at the beginning of this appendix). Terms like "Microsoft" or "Internet Explorer" must not be larger than the company, product, or site name. The logo should be reasonably spaced out from other text or graphics. It mustn't be altered in any way nor be resized. Finally, the following footnote must be included in the page: "Microsoft is a registered trademark and the Microsoft Internet Explorer Logo is a trademark of Microsoft."
- Terms: Internet Explorer should first be referenced as "Microsoft Internet Explorer 3.0." It then can be referenced as "Internet Explorer."
- Active link: Used in an Internet site, the logo must be an active link to the following address: http://www.microsoft.com/ie/ie.htm.

Static Logo Requirements

In order to be able to use the static logo, a Web site should provide at least one of the following requirements:

- Ratings: Support self-regulation of content to ensure appropriate access to your Internet site.
- Font formatting: Use font format tags that enable rich text presentation in Web pages.
- Marquees: Scroll text or graphics across your screen.

- Simple or enhanced tables: Use colors and textures to make tabular data more legible and visually appealing.
- Background sounds: Provide an auditory experience when your Internet site is accessed.
- Watermarks: Create a mark of distinction on your page.
- Inline AVIs: Graphically animate your page beyond static images.
- Simple or enhanced HTML frame tags: Simulate the appearance of a magazine with borderless, nonscrolling, floating frames, and even frames within frames.
- Enhanced HTML stylesheets: Control margins, line spacing, and placement of design elements; specify fonts and point sizes; get desktop publishing support for the Web.

Furthermore, the Web site must comply to the guidelines previously described.

Animated Logo Requirements

In order to be able to use the animated logo, the Web site must comply to the static logo requirements. It must also use at least one ActiveX-compatible control on the Web site. Microsoft also encourages the use of ActiveX compatible scripts.

How to Participate in the Program

To participate in the Internet Explorer Logo Program, register your Web site by filling out the registration form at http://www.microsoft.com/powered/pbbo.htm. Also have a look at the Microsoft Logo Program's complete (and latest) description at http://www.microsoft.com/ie/logo/.

Microsoft reserves the right to revoke the permission to use the logo for whatever reason. It also reserves the right to alter the guidelines of this program. However, it will try to warn people so that they can change their Web pages in order to comply with any new requirements.

Appendix C

Microsoft Visual C++ 4.1 and 4.2

Microsoft Visual C++ 4.x is a fully integrated development environment. This environment is also known as Microsoft Developer Studio (MDS). MDS provides a set of tools to complete your application in one place. These tools include text editor, resource editor, compiler, linker, debugger, and so on. This appendix provides an overview of the new features, tools, and ActiveX programming support within Versions 4.1 and 4.2.

Microsoft Visual C++ 4.1

Visual C++ 4.1 is a powerful and intuitive development environment that enables you to develop applications faster and better. It provides many reusable classes and wizards to help the software development. For example, Microsoft Foundation Classes (MFC), Controls Wizards, AppWizards, and Class Wizards.

An Integrated Environment That Simplifies the Project Cycle

Visual C++ 4.1 contains a fully integrated environment with the support of Source Control, Debugging, Info View, Class View, and File View. It also integrates the following packages:

- Microsoft Development Library
- Microsoft FORTRAN Powerstation
- Microsoft Visual Test
- Microsoft Visual SourceSafe
- Microsoft Visual C++ Cross-Development Edition for Macintosh

Visual C++ 4.1 has additional support for Windows 95 applications, OLE control containers, new common dialog boxes, and even World Wide Web access within the Developer Studio by choosing the Web Favorites command from the Help menu. The following sections discuss the detailed information on the new classes provided by MFC 4.1 and its samples.

The Microsoft Foundation Class Library is an *application framework* (often called framework) for writing applications for Microsoft Windows and other platforms that support the Win32 API. The framework is implemented as a group of C++ classes, many of which represent common objects such as windows, documents, views. and so on. MFC 4.1 provides many new classes. such as CHttpFilter to extend the functionalities of the Internet Information Server. These classes are named in the following list.

New MFC Classes

MFC has added five new classes, which are named in the following list, that encapsulate the functionalities provided by the ISAPI. By using these classes, you can create .DLLs to enhance the capabilities of the ISAPI-compliant Web server.

- CHtmlStream: Called by CHttpServer to send a HyperText Markup Language (HTML) stream back to the client.
- CHttpFilter: A class for creating an Internet server filter to filter messages sent to and from an

Internet server.

- CHttpFilterContext: A class that CHttpFilter uses to handle concurrent, multiple requests.
- CHttpServer: A class for creating an Internet server extension. Internet server extensions provide a more efficient, DLL-based alternative to Common Gateway Interface (CGI) applications.
- CHttpServerContext: A class that CHttpServer uses to handle concurrent, multiple requests.

Besides the ISAPI-related classes, classes supporting registry, Data Access Objects (DAO), and other functionalities are added. The following list is just an example of these classes.

- CDockState: a CObject class that can hold the state of a number of control bars. You can save the state to and load the state from the Registry, an INI file, or the binary contents of a CArchive object.
- CRecentFileList: a CObject class that supports the most recently used (MRU) file list.
- CSharedFile: Inherit from CMemFile. It supports RAM-based shared memory files for fast temporary storage and for transferring raw bytes or serialized objects between independent processes.

New Samples

Version 4.1 has added many new MFC samples that show you how to perform the following tasks:

- ACDUAL demonstrates custom interface and IDispatch. The sample implements the normal dispatch interface and then adds a custom interface derived from IDispatch to an MFC based OLE automation server.
- BINDSCRB demonstrates Document Object. This sample implements an OLE document object for use with the Binder tool in Microsoft Office 95 applications. This sample is based on the Scribble tutorial.
- DLGTEMPL dynamically create a dialog template and using the template with CreateDialogIndirect or InitModalIndirect.
- HTTPSVR demonstrates HTTP and WinSocket class. This example create a simple WWW HTTP server using MFC and WinSock classes.
- MFCUCASE demonstrates ISAPI classes and ISAPI extension wizards and how to write an ISAPI filter.
- ODBCINFO demonstrates DAO classes. This sample determines ODBC driver capabilities at runtime by opening a selected ODBC data source and displaying information using property pages.
- ROWLIST demonstrates list view control. This sample implements full-row selection in the report mode of the CListView control class.
- WWWQUOTE demonstrates ISAPI extension. This sample will retrieve stock quotes information and send back to the user.
- You can find these samples in the samples directory on the accompanying

The following section will list the ActiveX technology support within Version 4.x. Actually, most of these features are provided by Version 2.x except ISAPI Extensions.

Component Object Model Support

Version 4.1 supports Component Object Model (COM) extensively. If you would like your application to have COM support, one thing you must do is make sure that the project has OLE automation and OLE control enabled by clicking the OLE Automation and OLE Control check box shown in Figure C.1.

Figure C.1. Application with support of OLE automation and OLE control.

For more information on how to create an application that supports COM object, please refer to Chapter 3, "Creating COM Objects."

OLE Controls

OLE controls are very easy to create in Version 4.x. In Version 2.x, you have to install a separate control development package. In Version 4.x, the Control Wizard is built-in. You can select File|New, and then choose the Project Workspace in the New dialog box. You can see the Control Wizard in Figure C.2.

Figure C.2. Project Workspace: Wizard Type.

The OLE Control Wizard creates the OLE control framework for you. Then you can use Class Wizard to add custom methods, properties, and events. For more information on how to create OLE controls by using MFC, please refer to Chapter 5, "OLE Controls."

OLE Automation Server

You can use MFC to create an OLE automation server by using the OLE Control Wizard. The only thing you need to do is override IsInvokeAllowed() in your main control class.

OLE Control Container

Version 4.1 also supports Control Container. You can use the AppWizard to create the control container, which supports one or more controls.

ISAPI Extensions

Version 4.1 provides an ISAPI Extension Wizard to simplify the creation of the ISAPI DLLs. You can choose the ISAPI Extension Wizard to project workspace as shown in Figure C.3.

Figure C.3. ISAPI Extension Wizard in the Project Workspace.

For more information on how to use Version 4.x to create an ISAPI DLL, please refer to Chapter 14, "ISAPI Server Applications," and Chapter 15, "ISAPI Filter Objects."

Microsoft Visual C++ 4.2

Version 4.2 offered a lot of environment enhancements in addition to those found in Version 4.1. These enhancement are given here:

- C runtime library support including significant heap optimizations and standard C++ library.
- Architecture support for package partners such as Visual J++.
- Wizard support for new MFC functionalities, such as Document object support.
- Resource Editor support for data binding
- Resource Editor filters for JPEG and GIF

Besides the environment enhancement, Version 4.2 supports the additional ActiveX programming as discussed in the next section.

ActiveX Programming Support

In Visual C++ 4.2, the following features are supported by MFC:

- WinInet support: WinINet is the Win32 Internet Functions. It provides a programming interface for FTP, Gopher, and HTTP protocols. The developers do not need to worry about the basic socket layer programming and any protocol related details.
- Active Document support: The creation of document object is supported, but the hosting is not. With this feature, developers can create document objects.
- Asynchronous moniker support: With this feature, developers do not need to worry how to implement asynchronous behavior during the bind operation.
- Uniform Resource Locator (URL) moniker support: With this feature, developers can encapsulate the locating and download capabilities of the
- Open Database Connectivity (ODBC) support enhancements.

- OCX 96 support: The creation of OCX is supported, but not the hosting.
- Support for Data Binding.

To demonstrate the new features delivered by Version 4.2, the following new samples are contained under the sample directory.

New Samples

These new samples illustrate the use of several new MFC and Internet related features:

- COUNTER: Internet Server API (ISAPI) DLL
- FTPTREE: Internet FTP sample
- TEAR: Internet sample
- EXTBIND: Databound controls
- MDIBIND: Databound controls
- TESTHELP: OLE control with tooltips and help files
- VCTERM: Communication OLE control
- DLBCRB32: Toolbars in a dialog box
- DRAWPIC: Getting a picture from a CPictureHolder

Microsoft Visual C++ 4.x and Active Template Library

ATL stands for Active Template Library (ATL). It is a software package that assists developers in creating COM objects. ATL is also known as OLE COM AppWizard.

ATL is not built in Version 4.2. A copy of ATL can be found on the Microsoft Web http://www.microsoft.com/visualc/v42/atl/. After you download a copy of the ATL, you can install ATL by using the command pkunzip -d atl.zip. The include header file and lib will automatically be copied to the c:\msdev directory, and a template will be added to the Visual C++ Workspace. When you launch the Microsoft Visual C++ 4.x, you will see OLE COM AppWizard added in the Project Workspace shown in Figure C.4.

Figure C.4. OLE COM AppWizard.

Note

VC 4.2 does not have new features related to ATL. It also does not have a copy

of ATL 1.x or any other version.

The choice of your development platform is very important. I recommend that you choose Microsoft Visual C++ 4.2 rather than VC 4.1 because Version 4.2 provides a great deal of functionality that will greatly reduce the time involved in developing the same application under Version 4.1.

Appendix D

Visual J++

Visual J++ is the newest addition to Microsoft's series of Visual Development Tools. Though its name may not exactly have pleased some Java aficionados, the performance of the Visual J++ compiler has certainly brought Visual J++ in the limelight within the Java community. In this appendix we shall have a high-level overview of Visual J++, attempting to address basic as well as advanced issues.

Introduction

Java is a platform-independent, object-oriented, multithreaded and secure programming language. Java programs are compiled into a series of byte codes that are not machine specific. These byte codes are interpreted and executed within the Java Virtual Machine (VM) on a specific platform. The Java VM specification can be implemented on any platform, thus enabling the same Java programs to run on any platform. Microsoft Internet Explorer 3.0 provides a Java VM under 32-bit Windows platforms. Internet

Explorer also includes a Just-In-Time (JIT) compiler which compiles the byte codes into machine specific code achieving much faster execution of Java applets.

Java programs are typically developed in one of two forms:

1. 1. Applets: These are embedded in Web pages using the <APPLET> tag, and are downloaded and run by the browser on the user's local machine.
2. 2. Applications: These are standalone executables which can be executed independently of the browser.

Java ushers in a new paradigm for software development, and offers a vast potential than just developing fancy applets for use in Web pages. Java is similar to C++, and much more powerful than Visual Basic. It doesn't support multiple inheritance and pointer arithmetic, and offers a clean and robust environment including garbage collection and exception handling features.

ActiveX is a technology based on the Component Object Model (COM) foundation. COM is a foundation for component software, that specifies interfaces between component objects within an application or between applications, within a network or across networks (with Distributed COM, or DCOM).

There are innumerable software components supporting COM in the form of ActiveX Controls, Object Linking and Embedding (OLE) Automation Servers, which have been developed using popular tools like Visual C++, Visual Basic, and Borland's Delphi.

Using Visual J++, you can develop Java applets and programs that can make use of these software components. OLE Automation Objects such as those included within Microsoft Office and Lotus Notes, can be driven from within Java programs. High speed database access can be achieved using Data Access Objects (DAO) and Remote Data Objects (RDO).

You can also develop COM objects in Java, including ActiveX Controls. These Java programs can then be called locally or even remotely by other COM objects, which may be developed in a different programming language like C++. With built-in garbage collection and the ready functionality provided by the Microsoft Java VM implementation, Java makes COM programming simpler in several ways.

Thus Visual J++ gives developers a way to integrate the existing technologies with Java to make powerful Internet and Intranet applications, rather than forsake the ready functionality provided by widely available ActiveX software components.

Visual J++ Environment

Microsoft Visual J++ 1.0 is a development environment for creating Java applications. Visual J++ uses the Developer Studio interface familiar to Visual C++ developers. Developer Studio has a highly customizable interface with toolbars and dockable windows that can be moved, resized, hidden and arranged according to your preferences.

Your development work is focused in a Project Workspace which maintains the target options, build

settings, resources and source files that comprise your project. The Project Workspace View lets you switch between three views:

- The Class View gives a graphical representation of the classes in your source files, with a hierarchical tree displaying the member variables and functions of classes. Colored icons distinguish public, private and protected variables and functions. You can right click on any Class to add member variables, functions and classes, or go directly to the definition of the Class in the source file. Right clicking on any member function lets you set breakpoints at that function.
- The File View shows the files included in the current Project. You can open files for editing, compile individual files and set unique settings for building each file in the project by right clicking on any individual file.
- The Info View is the Information Viewer which gives you easy access to the complete online documentation including the User's Guide, Help files, the Java API reference and a comprehensive Java Class Library Hierarchy Chart. The InfoView Topic Window displays the relevant information with pop-up definitions for key terms and extensive links. You can search the entire online documentation for keywords.

Figure D.1 shows Visual J++ with the Introduction to Visual J++ InfoView Topic displayed.

Figure D.1. The Visual J++ environment.

Visual J++ comes with an *Applet Wizard* which provides a quick and easy way to create Java applets with pre-built support for multithreading, animation and event handling. The Applet Wizard can automatically create Java applets which can also run as standalone applications.

Let's develop a simple Applet using the Applet Wizard, before we look at other Visual J++ features. Later we shall have a glimpse of how we can pass information from an ActiveX Control to a Java applet embedded in the same HTML page.

Developing a simple applet using Applet Wizard

While you are in the Developer Studio, select File|New from the File menu and Developer Studio pops up a dialog box as shown in Figure D.2.

Figure D.2. The File New dialog box.

Select Project Workspace and press OK.

The New Project Workspace Dialog appears as shown in Figure D.3.

Figure D.3. The New Project Workspace dialog box.

Select Java Applet Wizard from the Type list and type myApplet in the Name text box. You can see the Java Virtual Machine as the Platform already selected for you. You can select the location of your project files by entering a valid pathname in the Location TextBox or by selecting an appropriate location using the Browse button. After you are done, press Create. This starts the Applet Wizard Steps sequence.

The Applet Wizard Step 1 is shown in Figure D.4.

Figure D.4. Applet Wizard Step 1.

You can make your applet support execution as a standalone application. The Applet Wizard can add standalone application functionality by providing an implementation for a main method within the applet derived class. The main method implemented in the applet derived class is ignored by a browser, whereas is the controlling point of execution for a standalone Java application. That is not all, however. Standalone applications do not by themselves have a graphical interface as applets in an HTML document. The Applet Wizard adds a graphical interface to the standalone implementation by adding a class derived from the Java Frame class. The code to handle these different states of execution is automatically provided by the Applet Wizard.

Note that myApplet defaults as the name of the desired class, and choose whether you wish to have explanatory and TODO comments added to the source code generated by the Applet Wizard. Click Next to proceed to Step 2.

You can request a sample HTML file with your embedded applet in Step 2 of the Applet Wizard as shown in Figure D.5.

Figure D.5. Applet Wizard Step 2.

Applet Wizard lets you specify an initial size for your applet. It adds an appropriate resize() call in the initialization method for the applet. Select the height to be 320 and width to be 100 pixels. Click Next to proceed to Step 3.

Step 3 offers powerful features that can be added to your applet including multithreading, animation, and event-handling support as shown in Figure D.6.

Figure D.6. Applet Wizard Step 3.

Applet Wizard can create a multithreaded applet which defines a Thread object in your applet class, initializing it during applet initialization. The code for starting and stopping the Thread object is automatically added. If you choose support for animation, Applet Wizard supplies default images for the animation which you can replace with your own. You can also create skeleton event handlers for events you wish to respond to, by selecting them from the list at the bottom.

It was never as easy to create a multithreaded Java Animation applet!

For our sample applet, we shall a use a simple multithreaded applet without animation and event handlers.

What's ahead? Click Next to go to Step 4 which appears as shown in Figure D.7.

Figure D.7. Applet Wizard Step 4.

An HTML author can pass parameters to the applet embedded inside the HTML page using <PARAM> tags in the HTML document. Applets can retrieve the parameter values using the getParameter() method of the Applet class. Applet Wizard automates this process by letting you define the applet parameters before it creates the applet.

Enter Message as the Name and Message to display as the Description of the parameter by clicking in the appropriate fields. Observe that Applet Wizard automatically assigns the String Type to the parameter and a member variable m_Message to hold the parameter value.

Applet Wizard automatically generates code to retrieve the parameter value in the m_Message private variable. It also provides an implementation for the getParameterInfo() method, where the Description field specified in Step 4 is returned by the method.

Click the Next button to go to Step 5 of the Wizard. This last step is where you add general information about the applet, as shown in Figure D.8.

Figure D.8. Applet Wizard Step 5.

The applet's getAppInfo() method returns the information specified here. This is where you declare to the world that you are the author of this applet, and the legal statements would usually follow.

When you click Finish, the Applet Wizard gives a summary outline of all choices you have opted for during the Wizard step sequence as shown in Figure D.9.

Figure D.9. Applet Wizard summary.

Click OK to generate your applet source file and a sample HTML file in the directory you specified. Developer Studio also generates the files for the Project Workspace and the Build and Compiler settings.

You can switch to Class View to see the myApplet class. Expand the myApplet class by clicking on the plus(+) sign next to it, and you can see a list of the methods and member variables in the myApplet class. Right click on the myApplet class and select Go To Definition from the pop-up menu. The Java source file myApplet.java opens in the source editor. Figure D.10 shows the Visual J++ environment with the Project Workspace View in Class View mode and the myApplet.java source file opened in the source editor on the right.

Figure D.10. Visual J++ Project Workspace Class View and Source Editor.

We shall add a public member variable message of type String to the myApplet class, which will contain a message that is displayed by the applet. Then we shall embed an ActiveX TextBox Control and a HTML button within our sample HTML page. When the HTML button is pressed, we shall retrieve the text in the ActiveX TextBox Control and pass it to our myApplet Java applet. Before we see how this is done, let us develop myApplet to display text from a String variable.

Right click on the myApplet class in Class View and select Add Member Variable. The Add Variable Dialog appears as shown in Figure D.11.

Figure 11. The Add Variable dialog box.

Enter String as the variable type, message as the variable name and select public from the drop-down list of Access Modifiers. As you type in the values and select the scope of the variable, the Add Variable Dialog shows the full declaration that will be added to our source file. Clicking OK adds the variable message with the following declaration in the myApplet class:

```
public String message;
```

Next, go to the definition of the myApplet paint() method by right-clicking on it and selecting Go To Definition in the Class View expanded list. Comment or delete the default g.drawString statement supplied by Applet Wizard and enter the following Java source code:

```
// myApplet Paint Handler

//--------------------------------------

 public void paint(Graphics g)

 {

    g.clearRect(0,0,319,99);

    g.drawRect(0,0,319,99);

    g.drawString(message,5,15);

 }
```

This is a simple paint() handler that clears the applet region, draws a rectangle and then displays the String in the message variable within the rectangle.

Now lets add a public method for updating the applet area when the button on the HTML page is clicked.

From the right-click pop-up menu of the myApplet class in Class View, select Add Method to bring the Add Method Dialog shown in Figure D.12.

Figure D.12. The Add Method dialog box.

Specify a return type of void for the public method named ShowMessage. Once again, observe the full declaration to be inserted in the source file at the bottom of the dialog.

You see the new method added to the list of methods under myApplet in the Class View. Add the following statement to the body of the method:

```
public void ShowMessage()

{

   repaint();
```

```
}
```

Add the following statement to the init() method to initialize our message variable to the value passed as a parameter:

```
message = m_Message;
```

Finally, comment out or remove the repaint() call in the run() method to prevent continuous updating of the applet area.

Save the file via File|Save from the File menu, and select Build myApplet from the Build menu. The Output window appears and displays the progress of the compilation.

Open the myApplet.html file and add "Default String" to the value attribute of the <PARAM> tag so that the applet tag is as shown below:

```
<applet

    code=myApplet.class

    id=myApplet

    width=320

    height=100 >

    <param name=Message value="Default String">

</applet>
```

Save the myApplet.html file by selecting File|Save from the File menu. Now you can run our applet by selecting Execute myApplet from the Build menu. Visual J++ fires up Internet Explorer 3.0 with the myApplet.html page displaying our myApplet Java applet.

Java Applet and ActiveX Control sample

You can open myApplet.html in ActiveX Control Pad to insert the ActiveX TextBox Control. For a detailed discussion of ActiveX Control Pad, refer to Chapter 11, "Using ActiveX Control Pad." Alternatively, you can add the following text directly in the myApplet.html file:

```
<OBJECT ID="TextBox1" WIDTH=292 HEIGHT=30

     CLASSID="CLSID:8BD21D10-EC42-11CE-9E0D-00AA006002F3">

        <PARAM NAME="VariousPropertyBits" VALUE="746604571">

        <PARAM NAME="Size" VALUE="6165;635">

        <PARAM NAME="Value" VALUE="Type your message here">

        <PARAM NAME="FontCharSet" VALUE="0">

        <PARAM NAME="FontPitchAndFamily" VALUE="2">

        <PARAM NAME="FontWeight" VALUE="0">

    </OBJECT>
```

Next, add a HTML Form button to myApplet.html like this:

```
<INPUT  TYPE=button VALUE="Show Message" NAME="ShowButton">
```

The connectivity between the Java applet and the ActiveX Control is achieved using VBScript. We shall add an event handler for the HTML button using VBScript. Add the following VBScript code to the myApplet.html file:

```
<SCRIPT language="VBScript">

<!--

 Sub ShowButton_OnClick

   document.myApplet.message = TextBox1.text

   document.myApplet.ShowMessage

 End sub

-->

</SCRIPT>
```

The Java applet is referenced from within VBScript by preceding the applet ID with the 'document.'

prefix. The above script sets the message public member variable of myApplet to the text of the TextBox1 ActiveX Control. It then calls the public method ShowMessage of myApplet to repaint the applet region on the HTML page.

The final HTML listing should now look like Listing D.1.

Listing D.1. The final myApplet.html listing.

```
<html>

<SCRIPT language="VBScript">

<!--

 Sub ShowButton_OnClick

   document.myApplet.message = TextBox1.text

   document.myApplet.ShowMessage

 End sub

-->

</SCRIPT>

<head>

<title>myApplet</title>

</head>

<body>

<hr>

<applet

    code=myApplet.class

    id=myApplet

    width=320

    height=100 >
```

```
    <param name=Message value="Default String">
</applet>
<P>
<INPUT  TYPE=button VALUE="Show Message" NAME="ShowButton">
<OBJECT ID="TextBox1" WIDTH=292 HEIGHT=30
     CLASSID="CLSID:8BD21D10-EC42-11CE-9E0D-00AA006002F3">
        <PARAM NAME="VariousPropertyBits" VALUE="746604571">
        <PARAM NAME="Size" VALUE="6165;635">
        <PARAM NAME="Value" VALUE="Type your message here">
        <PARAM NAME="FontCharSet" VALUE="0">
        <PARAM NAME="FontPitchAndFamily" VALUE="2">
        <PARAM NAME="FontWeight" VALUE="0">
    </OBJECT>
<hr>
<a href="myApplet.java">The source.</a>
</body>
</html>
```

Save the myApplet.html file and view it in Internet Explorer by either selecting Execute myApplet from the Build menu or pressing the Refresh button of the browser if it is still running. Enter any text into the ActiveX TextBox Control and click the ShowMessage button. The Java applet displays the text in response!

Figure D.13 shows the output in Internet Explorer with some sample text.

Figure D.13. Output of HTML page with Java applet and ActiveX Control.

Visual J++ Power Programming

Now that we have had a glimpse of the exciting potential of integration of Java applets with ActiveX Controls, here's a brief look at some features of Visual J++, which can put you in the forefront of the Java development momentum.

Visual J++ comes with a Resource Editor for editing resources such as menus, dialogs, toolbars and icons. You can design new resources from scratch or reuse ones you have previously developed for your Visual C++ projects. Resources are saved as templates which are then used in your Java programs using the Resource Wizard. Resources used in Java are limited to using features of the controls and components supported by the Java Abstract Window Toolkit (AWT).

The Resource Wizard is a very powerful tool to convert resource templates into compiled Java classes which can then be imported for use in your Java programs.

The Visual Debugger is a rich environment for debugging Java programs with disassembly, stepping and breakpoint support at the byte code level. You can watch variables, track changes with automatic highlighting, and change values while debugging. DataTips show the values of variables or expressions on which you linger the mouse pointer. You can also debug multiple applets running in the browser simultaneously in the Debugger.

Figure D.14 shows the myApplet Java applet being debugged in the Visual Debugger.

Figure D.14. Visual Debugging of myApplet with byte code disassembly.

The figure shows the byte code disassembly in the main window, the Variables window at the bottom left and the Watch window at the bottom right. You can control execution from the Debug Toolbar. You can even Step Out of a method!

Source-Code Control with Visual SourceSafe is integrated within Developer Studio. You can work seamlessly with your Visual SourceSafe Database as the SourceSafe commands are available from right within the Visual J++ development environment.

The Graphics Editor lets you create, edit and save GIF and JPEG graphic files for use with your Java programs or HTML pages. It comes with an assortment of painting tools on a palette much like those in standard painting programs.

Remote Data Objects (RDO) and Data Access Objects (DAO) provide high-speed database access to your Java applications. RDO enables you to create and control components on a remote database. Thus Visual J++ gives you instant access to most existing databases using the ubiquitous Open Database Connectivity (ODBC) interface.

Third Party Products shipped along with the Visual J++ 1.0 CD include: the Java Generic Library (JGL) from ObjectSpace Inc., the premier Java library for algorithms and container libraries; Liquid Motion from DimensionX; Jamba from Aimtech Corporation, tools for multimedia-enabling Web sites; and Mail Wizard from Neural Applications Corporation for adding Internet mailing capabilities to your Java applications.

Using Java with Component Object Model (COM) Software Objects

The most exciting aspect of the Java Virtual Machine implementation and the Visual J++ compiler is the symbiotic nature of two technologies that have evolved independently of each other. We have seen how Java applets can be scripted by VBScript, and how the Java VM in conjunction with VBScript can enable ActiveX Controls to communicate with a Java applet.

The Java Virtual Machine exposes a Java applet as an ActiveX Control to VBScript and other COM Objects. This means that Java applets can be used by any application that makes use of ActiveX Controls such as Visual Basic, Delphi, Access and PowerBuilder applications.

What about the other way around?

Using ActiveX Controls and COM Objects from Java

It turns out that you *can* use an ActiveX Control from within a Java applet that resides on the same HTML page. The HTML file can pass the ActiveX Control as a parameter to the Java applet. However, the COM interfaces exposed by ActiveX Controls are not useable by Java programs directly. The COM objects and interfaces that are described in the Object's Type Library have to be converted to Java classes that enable Java programs to access them as if they were Java classes.

Visual J++ comes with the Java Type Library Wizard which automates this process. The Type Library Wizard lets you create Java classes from the type library information for various COM objects stored in the registry. Type Libraries store information about the programming model of a COM component such as its classes, interfaces etc. In order to use those COM objects in Java, the type library information is converted into Java classes, which can then be imported and used by Java programs. These special class files are distinguished from normal Java class files by a special tag recognized by the Java VM.

The Type Library Wizard invoked from the Tools menu is shown in Figure D.15.

Figure D.15. The Visual J++ Java Type Library Wizard.

The Type Library Wizard lists all the Type Libraries installed on your system with a blank check box beside each. You can select the Type Library that describes the COM class you wish to use in your Java program. When you click OK, the Type Library Wizard generates Java Class files which act as wrappers for the COM classes.

If you are aware of the security restrictions placed on downloadable Java applets, you must be wondering how a Java applet can make a direct call into an ActiveX Control or a COM Object. Java applets run within a restricted environment that isolates it from the system to prevent any harm. A Java applet using

a COM Object's functionality accesses resources outside this restricted environment. The security issues are resolved by distinguishing all COM classes as either *trusted* or *untrusted*.

Classes loaded from the local CLASSPATH or digitally signed are considered as trusted and are permitted to use COM services. All other classes such as those downloaded from a network, are considered untrusted and are restricted to their secure execution environment, referred to as a sandbox. If you wish to create trusted applets which can access COM services, you have to package your classes in a digitally signed .CAB (cabinet) file. The Cabinet Development Kit and the Code Signing Kit are included with Visual J++.

If you are using COM objects from Java, the OLE Object View tool can come especially handy.

The OLE Object View Tool is a powerful utility which lets you inspect the details of the OLE objects installed on your system. Developers of OLE/COM objects can use the OLE Object View Tool to test their OLE Objects. If you are using a programmable control or Automation Server from your Java programs, OLE Object View can display method attributes and parameter names for all the OLE Automation Servers installed on your system. This proves indispensable if you do not have comprehensive or accurate documentation of the interfaces exposed by the OLE Server. Figure D.16 shows the main screen of the OLE Object View Tool. (The OLE Object View Output may vary from system to system depending on which OLE Objects are installed on a particular system).

Figure D.16. OLE Object View Tool main screen.

The OLE 2.0 Objects installed on your system are displayed on the list at the left, with different icons distinguishing OLE Objects with differing functionality such as OLE Automation Servers, OLE Controls, insertable and container objects, Type Libraries, etc. The OLE Object View also includes a Type Library Browser shown in Figure D.17.

Figure D.17. OLE Object View Type Library browser.

If you develop a Java applet using COM services on an HTML page, you have to ensure that the user's version of Internet Explorer fully supports Java and COM integration. You should specify the following tag on the HTML page containing your applet:

```
<OBJECT

CLASSID="clsid:08B0E5C0-4FCB-11CF-AAA5-00401C608500"

CODEBASE="http://www.microsoft.com/java/IE30Java.cab#Version=1,0,0,1">

</OBJECT>
```

If the Java support in the version of Internet Explorer is outdated, Internet Explorer automatically downloads and installs the updated version.

Developing COM Objects in Java

The Java VM facilitates development of COM objects in Java since it automatically provides an implementation of the base COM Interface IUnknown. Reference counting, a mechanism with which an object keeps track of how many clients are using it, is automatically handled by the garbage collection in Java. This makes development of COM Objects in Java easier in several respects in contrast with C++. We shall have a brief overview of the steps required to implement a COM object in Java. You can find comprehensive details in Visual J++ Books Online.

Developing COM Objects in Java requires the following steps:

1. 1. Describing the interfaces exposed by the COM class in Object Description Language (ODL).
2. 2. Using MkTypLib or the Microsoft Interface Definition Language (MIDL) compiler to generate a Type Library from the ODL
3. 3. Using JavaTLB command-line Tool to generate Java class wrapper files from the Type Library
4. 4. Implementing the Java Class in .java source files
5. 5. Using the JavaReg tool to register the Java class as a COM class. The JavaReg tool creates special registry entries for COM objects implemented in Java. This ensures that when COM services implemented in Java are requested, the Java VM can be launched to load the appropriate Java class.

Note that the COM client is not aware of the language in which the particular COM service is implemented. Also note that if you distribute your COM object developed in Java, you need to provide a way to register your COM class on the user's system.

Conclusion

Visual J++ 1.0 is an impressive environment in its first version. It is the only development environment today that lets you take advantage of existing ActiveX Components and offers a rich set of features that make your Java development efforts highly productive and efficient.

Appendix E

ActiveX Template Library

The ActiveX Template Library (ATL) was created to simplify the programming of COM objects. ATL enables you to focus on programming the functionality of your objects

This appendix defines what ATL is and when ATL should be used in your object development. In addition, this appendix outlines the software components contained within the ATL product.

What Is ATL?

ATL is a set of template-based C++ classes that enables developers to create fast, lightweight COM objects. Developers can define their own types of data that are specific to their own applications by using the template-based classes.

ATL simplifies object type definitions by the following:

- Eliminating the need for static libraries (LIBs) or dynamic link libraries (DLLs)
- Eliminating the need for C runtime library startup code
- Implementing tear-off interfaces

What Object Model Does ATL Support?

ATL supports the key OLE COM (Component Object Model) features such as aggregation, dual interface, connection points, enumerations, and tear-off interfaces.

ATL also supplies a custom AppWizard, which can be used with Visual C++ 4.1 and later to create a basic skeleton of a COM object. This wizard is targeted for creating dual interfaces in particular. ATL can be used to create any arbitrary COM interface, however. The following sections discuss the COM Server interfaces and the threading model ATL supports.

COM Servers

The ATL supports the following COM servers:

- Inprocess servers
- Local servers
- Remote servers using the Distributed Component Object Model (DCOM) or Remote Automation (RA)
- Aggregatable servers

Threading Model

The threading models ATL supports include No threading, Apartment-model threading, and free threading.

Interface Types

ATL supports many interface types, such as the following:

- Custom COM interfaces
- Dual interfaces
- IDispatch interfaces

Other Interfaces

ATL also supports interfaces such as these:

- Enumerations (IEnumXXX)
- Connection points
- OLE error mechanism (IErrorInfo)

Using ATL with C++ Frameworks?

The Microsoft File Class (MFC) is a C++ framework that contains standard, customizable implementations of the major OLE and ActiveX interfaces and the logic around their use. This makes it fairly simple to add, for example, the capability of hosting ActiveX controls in an MFC application, or to create an application that enables other applications to embed objects within it.

For creating advanced OLE- and ActiveX-based applications, such as ActiveX controls, ActiveX Documents, and regular embedded and linked objects, MFC is an excellent C++ framework.

To build them with ATL requires a significant amount of work, for relatively little gain.

ATL

ATL is focused primarily for creating COM servers that have specific requirements of size and speed. ATL is the best choice to create generic COM objects for OLE automation objects with dual interface support.

If your intent is to support the COM free threading model, which is available only under Windows NT 4.0 or later versions of Windows, ATL should be used.

ATL Software Product

ATL can be downloaded from http://www.microsoft.com/visualc/v42/atl/default.htm. The ATL is distributed in the form of a zip file that includes the following:

- Documentation ATL white paper (ATLWHITE.TXT) and ATL general information (ATLTN001.TXT)
- The ATL files: ATLBASE.H, ATLCOM.H, ATLIMPL.CPP, and ATLUTIL.H
- OLE COM AppWizard for generating starter apps (atlwiz.awx)
- Two samples (Beeper and Labrador) that show how to use ATL

What Do I Need to Make Use of ATL?

ATL is contained in Visual C++ 4.*x* and the Win32 SDK products. Purchasing either of these products will give you the libraries and supportive documentation you need to get started.

Note

ATL can also be used with the NT 4.0 beta SDK component. The advantage of using the NT beta 4.0 beta version of the MIDL compiler is that you no longer have to maintain an ODL file in addition to your IDL file. For more information on the new MIDL compiler, see the NT 4.0 beta documentation.

How Do I Build an ATL Server?

ATL includes a custom AppWizard you can use to build a basic skeleton of your server application.

The following steps are necessary to create a new project with the custom AppWizard:

1. 1. From the File menu, choose New, and then select Project Workspace.
2. 2. Choose OLE COM wizard in the New Project Workspace dialog box.
3. 3. Click the Finish button.
4. 4. Use the Custom Build tab in the Project Settings dialog box to add the necessary custom build rules.
5. 5. Add methods to your object's interface by modifying the IDL or ODL files if using older tools.
6. 6. Modify the object's H file and specify the method there.
7. 7. Modify the object's CPP file to specify the implementation of each method.
8. 8. Build or rebuild all after having added all your methods.

Base Classes Provided with ATL

The next sections introduce some basic classes provided by ATL.

CComObjectRoot

All ATL COM objects are usually derived from the CComObjectRoot or CComObjectBase classes. CComObjectBase is inherited from CComObjectRoot.

CComISupportErrorInfoImpl

CComISupportErrorInfoImpl is inherited from ISupportErrorInfo.

ATL supports the OLE error reporting mechanism with the Error() member function in CComObjectBase and the CComISupportErrorInfoImpl classes. These classes each have a member, InterfaceSupportsErrorInfo(), which indicates that returning rich error information is supported.

CComTearOffObjectBase

Tear-off interfaces can be used for interfaces to an object that are likely to be used only rarely, or for groups of related interfaces. They enable some interfaces to be implemented in another object. This method saves four bytes that would be required for a virtual table (vtable) pointer when an object supports interfaces through multiple inheritance instead.

To do this, you declare a class that inherits from all the interfaces you want to implement in the tear-off interface and from CComTearOffObjectBase<CLSID*, class *Owner*>, where *Owner* is the class of the main object:

```
class CComTearOffObjectBase : public CComObjectBase<pclsid>
```

CComConnectionPoint

ATL supports the COM connection point mechanism with the CComConnectionPoint and CComConnectionPointContainerImpl classes.

The following gives the relationship between the classes:

```
class CComConnectionPointBase : public IConnectionPoint

class CComConnectionPoint : public CComConnectionPointBase

class CComConnectionPointContainerImpl : public IConnectionPointContainer
```

CComEnumImpl

ATL supports OLE automation collections with the CComEnumImpl class. This class enables you to specify whether you enumerate on the actual data or a copy of the data. You need to specify a class that provides the copy, init, and destroy member functions:

```
class CComEnumImpl : public Base
```

Note

The Base class is the user's class, which derives from CComObjectRoot and whatever interfaces the user wants to support on the object.

ATL is great tool that developers can use to create generic COM objects for OLE automation objects with dual interface support.

Appendix F

HTML Enhancement by Internet Explorer 3.0

by Weiying Chen

IE 3.0 supports a number of HTML enhancements, which include frames, style sheets, tables with background colors or image settings for table cells, the OBJECT tag to insert ActiveX controls, and animated .GIF and .BMP images. In particular, frame support includes bordered frames, borderless frames, and targeted window and floating frames. All these enhancements will be examined in detail in this appendix, along with examples.

Border and Borderless Frames

Independent frames (windows) can be defined in a Web page by using frame tags. The frame tags include FRAME, FRAMESET, NOFRAMES. FRAME specifies a single frame within a FRAMESET. There is no matching tag for FRAME. FRAMESET describes how frames are laid out on a page. NOFRAMES indicates that the content enclosed between a NOFRAMES tag can be viewed by browsers that do not support frames.

By default, FRAMESET hosts each frame with borders. The page shown in Figure F.1 consists of two frames.

Figure F.1. Bordered frames.

Left frame provides a table, with two items in the table cell, Step 1 of 2 and Step 2 of 2. The cells are hyperlinked. When Step 1 of 2 is clicked, the content of the right frame will be changed to the OLE COM AppWizard. Step 1 of 2 is shown in Figure F.2.

Figure F.2. Link: Step 1 of 2 is clicked.

When Step 2 of 2 is clicked, the content of the right frame will be changed to the OLE COM AppWizard. Step 2 of 2 is shown in Figure F.3.

Figure F.3. Link: Step 2 of 2 is clicked.

Listing F.1 highlights the implementation of the page displayed in Figure F.1.

Listing F.1. Complete code listing for page in Figure F.1.
f1.htm

```
<HTML>

<HEAD>

    <TITLE>Bordered Frames with scrolling bar</TITLE>

</HEAD>

<FRAMESET COLS = "250, *" FRAMEBORDER=1 FRAMESPACING=3>

    <FRAME NAME = "index" MARGINWIDTH=5 MARGINHEIGHT=5

           SCROLLING="yes" SRC="f2.htm">

    <FRAME NAME = "main" MARGINWIDTH=10 MARGINHEIGHT =10

           SCROLLING ="yes" SRC="f3.htm">

</FRAMESET>

<NOFRAMES>

Your browser doesn't support frames.

Click <A HREF="noframes.htm"> here </A> to go to the non-frames page.

</NOFRAMES>

</HTML>

<HTML>

<STYLE>

    BODY {font: 11pt/12pt Arial; color: brown}
```

```
    P  {font: 11pt/14pt Times; color: blue; text-indent: 0.25in; margin-right:
0.25in}

    H2  {font: 12pt Arial; color: Yellow}

</STYLE>

<BODY>

<H2> OLE COM AppWizard Screen Menu</H2>

<TABLE ALIGN=left CELLSPACING=5>

<TR>

<TD ALIGN=CENTER VALIGN=MIDDLE BGCOLOR=#ddddff WIDTH=130 HEIGHT=30>

<A HREF="index1.htm" TARGET="main">Step 1 of 2</A>

</TD>

</TR>

<TR>

<TD ALIGN=CENTER VALIGN=MIDDLE BGCOLOR=#ddddff WIDTH=130 HEIGHT=30>

<A HREF="index2.htm" TARGET="main">Step 2 of 2</SPAN></A>

</TD>

</TR>

</TABLE>

</BODY>

</HTML>
```

f3.htm

```
<HTML>

<HEAD>

<TITLE>OLE COM AppWizard</TITLE>

<STYLE>
```

```
    BODY {font: 9pt/10pt Arial; color: white}
    P  {font: 13pt/14pt Times; color: green; text-indent:
        0.25in; margin-right: 0.25in}
    H1  {font: 30pt Times; color: brown}
    H2  {font: 18pt Arial; color: Green; font-style: italic}
</STYLE>
</HEAD>
<BODY>
<H2> Active Template Library</H2>
<P>The Active Template Library<SPAN STYLE="background: white">
(ATL)</SPAN>was created to simplify the programming of
<SPAN STYLE="background: white"> COM objects</SPAN>.
It includes a Custom AppWizard you can use to build a basic
skeleton of COM server.When the OLE COM AppWizard option is
selected in the Project Workspace, the dialog will
be displayed</P>.
</BODY>
</HTML>
```

INDEX1.HTM

```
<HTML>
<IMG SRC="http://www.sample.com/f1.bmp">
</HMTL>
```

INDEX2.HTM

```
<HTML>
<IMG SRC="http://www.sample.com/f2.bmp">
```

```
</HTML>
```

Note that Listing F.1 is included on the CD.

In Listing F.1, the HTML tag indicates that this file is an HTML document.

The HEAD tag specifies the HTML document heading.

The TITLE tag specifies the HTML document title displayed in the browser's title bar.

FRAMESET defines the frame's layout in a page. The COLS attribute indicates that the page is divided into columns. In this example, the page is divided into two columns, showing two files f2.htm and f3.htm. ROWS can be used instead of COLS if the page is divided into rows. The 250 in "250, *" indicates that the first column is 250 pixels wide, * indicates "whatever is left over."

Besides using pixels to indicate the column width, the percentage of screen width can be used; for instance, instead of using "250, *", "25%, *" can be used. 25% means 25% of the screen width for the first column.

The FRAMEBORDER attribute provides an option to display or not to display the frame border. 1 means border, 0 means borderless. The default value for the FRAMEBORDER attribute is 1, which means the framesets have borders between the frames. If the FRAMEBORDER is set to 0, the framesets will have borderless frames with the frames right against each other. Borderless frames provide a seamless look for the page. The Borderless feature is only supported by IE 3.0.

The FRAMESPACING attribute indicates that there is an additional space between frames in pixels.

The FRAME tag defines a single frame in a frameset. The attribute NAME specifies a name for the frame. MARGINWIDTH specifies in pixels a margin width for the frame. MARGINHEIGHT specifies in pixels a margin height for the frame. The SCROLLING attribute indicates a scrolling frame. The SRC attribute specifies a source file for the frame. FRAME does not have a matching tag.

From Listing F.1, observant readers notice the following code:

```
<A HREF="index1.htm" TARGET="main">Step 1 of 2</A>
```

Tag A refers to Anchor. Attribute HREF refers to creating a hyperlink. TARGET specifies loading the link specified by HREF into the target window. In this case, the value of the target window is a frame named main. The value of the target window can also be _blank, _parent, _self, or _top. _blank refers to loading the link into a new blank window. _parent means loading the link into the parent of the document where the link is. self means loading the link into the window where the link is clicked. _top refers to loading the link into the full body of the window.

NOFRAMES tag indicates that the content between the beginning and its matching tag can be viewed by the browsers that do not support frames, whereas the content will be ignored by the browsers that do support frames. This tag can be used so that a page can be created for both types of browsers.

IE 3.0 also supports .BMP and animated .GIF images. The following code from Listing F.1 demonstrates how to use the IMG tag.

```
<IMG SRC="http://www.sample.com/f1.bmp">
```

Tag IMG indicates the insertion of a graphics file, which includes .BMP or .GIF. Attribute SRC indicates the URL where the graphic file locates.

Besides the border and borderless frames, floating frames are a new feature supported by IE 3.0.

Floating Frames

Floating frames enables you to place any HTML page where an image can be placed in other browsers. Floating frames are a separate browser. Listing F.2 demonstrates how to create a floating frame.

Listing F.2. Floating frame (f4.htm)

```
<HTML>

<BODY>

<IFRAME WIDTH=300 HEIGHT=300 ALIGN=RIGHT

HSPACE=20 VSPACE=20  SCROLLING=YES FRAMEBORDER=1

SRC="http://www.microsoft.com/vbscript">

    <FRAME WIDTH=300 HEIGHT=300 ALIGN=RIGHT

    HSPACE=20 VSPACE=20  SCROLLING=YES FRAMEBORDER=1

    SRC="http://www.microsoft.com/vbscript">

</BODY>

</IFRAME>
```

This page will display a 300 by 300 pixel window on your page. You can see this on http://www.microsoft.com/vbscript page.

In Listing F.2, IFRAME tag specifies the floating frame. Attribute WIDTH indicates the window width in pixels. HEIGHT indicates the window height in pixels. ALIGN the window to left or right. SCROLLING specifies whether the window has scroll bars. FRAMEBORDER indicates whether the window has a border.

Style Sheets

Style Sheet is another new feature supported by IE 3.0. By using this feature, layout and formatting information can be added to the Web page, such as specifying page margins, font size, and indent and line spacing. Figure F.1 shows the effect of Style Sheet.

From Listing F.1, notice the following code:

```
<STYLE>
    BODY {font: 11pt/12pt Arial; color: brown}
    P  {font: 11pt/14pt Times; color: blue; text-indent: 0.25in;
        margin-right: 0.25in}
    H2  {font: 12pt Arial; color: Yellow}
</STYLE>
```

Tag STYLE provides the custom rendering information, which overrides the client default style sheets.

```
P  {font: 11pt/14pt Times; color: blue; text-indent: 0.25in;
   margin-right: 0.25in}
```

Attribute font groups the font-family, font-size, line-height, font-weight, and font-style together, as listed in Table F.1. The font-weight and font-style must be specified before other font attributes.

Table F.1. Font attributes.

Attribute Value Meaning Usage font-family Courier, Times[el] Font family name Font-family: Times Font family name font-size pt(points) font size Font-size: 11pt cm(centimeters) in(inches) px(pixels) line-height pt(points) Line height from line-height: 14pt the base line cm(centimeters) in(inches) px(pixels) %(percentage) font-weight Extra-light Font thickness Font-weight: bold light demi-light medium demi-bold bold extra-bold font-style Normal Font style Font-style: italic italic

```
{font: 11pt/14pt Times; color: blue;
```

can be expanded into

```
P {font-family: Times;
    font-size: 11pt;
    line-height: 14pt};
```

The color attribute sets the named color, such as blue, red, or an RGB value. The text-indent attribute specifies text indentation in points, inches, pixels, or centimeters. 0.25in means the paragraph will be indented 0.25 of an inch from the left margin. The margin-right attribute sets the distance from the right edge of the page. Besides margin-right, there are other margin attributes such as margin-left and margin-top. margin-left sets the left edge of

the page, whereas margin-top sets the top edge of the page. These three attributes can be combined into the margin attribute.

The following line is from f3.htm:

```
<SPAN STYLE="background: white">(ATL)</SPAN>
```

Tag SPAN indicates that the localized style information will be applied to the enclosed text. The STYLE attribute indicates the local style. The background attribute specifies a background color (white) to highlight the enclosed text (ATL). Besides specifying a background color, a background image can be specified with the following syntax:

```
{background: URL(http://www.sample.com/some.gif)}.
```

Note, the URL is placed between ().

For more information on Style Sheets, please refer to the Style Sheet Guide at http://www.microsoft.com/workshop/author/htmlhelp/.

Table Enhancements

IE 3.0 supports several new table features, such as background images placed in individual table cells, borders (rules) between rows or columns, or aligning text by a baseline.

The left frame in Figure F.1 gives an example of setting in individual table cells a blue background color.

Observant readers notice the following lines from Listing F.1:

```
<TABLE ALIGN=left CELLSPACING=5>

<TR>

<TD ALIGN=CENTER VALIGN=MIDDLE BGCOLOR=#ddddff WIDTH=130 HEIGHT=30>

<A HREF="index1.htm" TARGET="main">Step 1 of 2</A>

</TD>

</TR>
```

In this code, tag TABLE indicates creation of a table. The ALIGN attribute specifies the table alignment; it could be left or right. The CELLSPACING attribute specifies in pixels the amount of space between the exterior of the table and the cells in the table.

Tag TR indicates creation of a row in a table.

Tag TD creates a cell in a table. The ALIGN attribute specifies the horizontal alignment of text in a cell. The default value is center. Its value can also be left or right. The VALIGN attribute specifies the vertical alignment of text in a cell. The default value is middle. Its value can also be top, bottom, and baseline. top means text is aligned with the top of each cell. button means text is aligned with the button of each cell. baseline means text in the same row in other cells is aligned along a common baseline. This is a new table feature supported by IE 3.0. The BGCOLOR attribute specifies a background color.

Besides setting the background color, the border between rows or columns can be specified by using

```
<TABLE RULES=ROWS>
```

or

```
<TABLE RULES=COLS>
```

Object Support

One of the major features provided by IE 3.0 is to support the use of ActiveX controls. The OBJECT tag is defined to indicate the insertion of the control in a page. Listing F.3 demonstrates how to use the OBJECT tag.

Listing F.3. Stock ticker control (f5.htm).

```
<HTML>
<OBJECT ID="stocktick"
    CLASSID="clsid:CB005660-D0C7-11cf-B7F6-00AA00A3F278"
    CODEBASE="http://207.68.151.220/investor.cab#ver%3d4,70,0,1115"
    ALIGN="baseline"
    BORDER="1"
    WIDTH="400"
    HEIGHT="34"
    TYPE="application/x-oleobject">
    <PARAM NAME="NumLines" VALUE="2">
    <PARAM NAME="InvestorURL"  VALUE="http://investor.msn.com/">
```

```
    <PARAM NAME="DataObjectActive" VALUE="1">

    <PARAM NAME="ServerRoot"  VALUE="http://investor.msn.com">

    <PARAM NAME="Keywords"  VALUE="http://www.msnbc.com/news/ticker.txt">

</OBJECT>

</HTML>
```

In Listing F.3, OBJECT tag inserts an object into the HTML document. The object can be OLE controls, a Java Applet, or images and documents.

The ID attribute designates a name for this object.

CLASSID specifies the object implementation. For an OLE control, the CLSID of the control has to be provided. CB005660-D0C7-11cf-B7F6-00AA00A3F278 is the string representation of the CLSID associated with the control.

The CODEBASE attribute identifies the code base for this object. It includes an http address and the version number.

The ALIGN attribute sets the alignment of the object. The value can be baseline, center, left, middle, right, textbottom, textmiddle, and texttop.

The BORDER attribute specifies the width of the border. The WIDTH attribute specifies the width for the object. The HEIGHT attribute specifies the height of the object.

The TYPE attribute specifies the internet media type (MIME) for the data.

Tag PARAM sets the object's value. The NAME attribute specifies the property name exposed by the object. The VALUE attribute specifies the property value.

When f5.htm is viewed in the IE 3.0, the content in Figure F.4 will be displayed.

Figure F.4. Stock ticker control in IE 3.0.